SIAM J. COMPUT.
Vol. 23, No. 1, pp. 1–11, February 1994

COMPUTATIONAL COMPLEXITY OF SPARSE RATIONAL INTERPOLATION *

DIMA GRIGORIEV†, MAREK KARPINSKI‡, AND MICHAEL F. SINGER§

Abstract. The authors analyze the computational complexity of sparse rational interpolation, and give the first deterministic algorithm for this problem with singly exponential bounds on the number of arithmetic operations.

Key words. computational complexity, interpolation, sparse rational functions, arithmetic operations

AMS subject classifications. 68Q25, 68Q40, 68Q15, 26C15

Introduction. Given a black box to evaluate a t-sparse (a quotient of two t-sparse polynomials) n-variable rational function f with integer coefficients, we can find the coefficients and exponents appearing in a t-sparse representation of f using $(t^{(nt)} \log d)^{O(1)}$ black box evaluations and arithmetic operations, and with arithmetic depth $(nt \log d)^{O(1)}$, where d denotes the degree of t-sparse representation of f (see the theorem at the end of §4 for an exact statement of this result). Although these bounds involve the size of exponents, this dependency arises only at the end of our algorithm. The algorithm genuinely produces (i.e., produces in a way whose arithmetic complexity does not depend on the size of the coefficients of f or on the degree of f, [19]) a polynomial whose roots are p-powers (for some small p) of the exponents appearing in a t-sparse representation of f. All known algorithms to find the roots of this polynomial (even knowing that they are p-powers) have complexities that depend on the size of the roots. This dependency also occurs in algorithms for interpolating t-sparse polynomials (cf., [1]) for the same reason.

To find the exponents appearing in some t-sparse representation of a t-sparse univariate rational function $f(X)$, we proceed as follows: We consider representations of $f(X)$ of the form $(\sum_{i=1}^{t} a_i X^{\alpha_i})/(\sum_{i=1}^{t} b_i X^{\beta_i})$, where a_i, b_i, α_i, β_i are real numbers. Such a function is called a *real quasi-rational function.* Furthermore, we call such a representation minimal if it has a minimal number of nonzero terms in the numerator and denominator; it is called normalized if some term is 1. We show that there are only a finite number of minimal normalized representations and that the exponents must be integers. We are able to produce a system T of polynomial equalities and inequalities (whose coefficients depend on the values of $f(X)$ at $t^{O(t)}$ points) that determine all the possible values of any such α_i and β_i. Using the methods of [13], we can then find all α_i and β_i. To find the exponents when $f(X_1, \dots, X_n)$ is a multivariate polynomial, we show how to produce sufficiently many n-tuples of integers $(\nu_1, \dots, \nu_n)$ such that the exponents of f can be recovered from the exponents of all the $f(X^{\nu_1}, \dots, X^{\nu_n})$.

Complexity issues for t-sparse polynomial and rational function interpolation have been dealt with in several papers. Polynomial (black box) interpolation was studied in [1], [2], [9], [12], [17], [19], [27], and [28]. For bounded degree rational interpolation (when the bound on the degree is part of the input), see [15], [16], [25]. Approximative unbound interpolation also arises naturally in issues of computational learnability of sparse rational functions

*Received by the editors January 15, 1991; accepted for publication (in revised form) May 19, 1992. A preliminary version of this paper appeared in [10].

†Department of Computer Science, University of Bonn, 5300 Bonn 1, Germany, and Steklov Mathematical Institute, Fontanka 27, St. Petersburg, 191011 Russia. The author would like to thank the Max Planck Institute in Bonn for its hospitality and support during the preparation of this paper.

‡Department of Computer Science, University of Bonn, 5300 Bonn 1, Germany and International Computer Science Institute, Berkeley, California 97420. The work of this author was supported in part by Leibniz Center for Research in Computer Science, by the Deutsche Forschungemeinschaft grant KA 673/4-1.

§Department of Mathematics, North Carolina State University, Raleigh, North Carolina, 27695-8205. This author would like to thank the University of Bonn for its hospitality and support during the preparation of this paper.

(cf. [21]). The present authors have previously studied the problem of interpolation of rational functions in [10], but the algorithm presented there for finding the exponents had considerably worse complexity. The present paper significantly improves the results of that paper by introducing the notion of a minimal representation (allowing us to compute directly a finite set of possible exponents instead of just bounding them) and a new technique for reducing multivariate interpolation to univariate interpolation. As we shall see, these ideas give us a more efficient algorithm.

The rest of the paper is organized as follows: In §1 we give formal definitions of a quasi-rational function and related concepts and prove some basic facts about these functions. In §2 we introduce some useful linear operators on fields of these functions. We use these operators to derive criteria for a function to be t-sparse. In §3 we use these criteria to give an algorithm for t-sparse univariate interpolation. In §4 we again use these operators to show how multivariate interpolation can be reduced to univariate interpolation. Complexity analyses of the algorithms are also given in §§3 and 4.

1. Quasi-rational functions.

A finite sum

$$\sum_I c_I \mathbf{X}^I, \tag{1}$$

where $I = (\alpha_1, \dots, \alpha_n), \alpha_i \in \mathbb{C}$, $X^I = X^{\alpha_1} \cdot \dots \cdot X^{\alpha_n}$, $C_I \in \mathbb{C}$ is called a *quasipolynomial* of n variables. The set of quasipolynomials forms a ring under the obvious operations and we denote this ring by $\mathbb{C}\langle X_1, \dots, X_n\rangle$. The subring of quasipolynomials (1) with $\alpha_i \in \mathbb{R}$ and $c_I \in \mathbb{R}$ will be referred to as the ring of *real quasipolynomials* and will be denoted by $\mathbb{R}\langle X_1, \dots, X_n\rangle$. A ratio of two quasipolynomials (real quasipolynomials) is called a *quasi-rational function* (*real quasi-rational function*). The set of such functions forms a field that we denote by $\mathbb{C}\langle\langle X_1, \dots, X_n\rangle\rangle(\mathbb{R}\langle\langle X_1, \dots, X_n\rangle\rangle)$. Note that $\mathbb{Q}(X_1, \dots, X_n) \subset \mathbb{R}\langle\langle X_1, \dots, X_n\rangle\rangle$. We use the expressions *polynomial* or *rational function* in the usual sense (that is, for a quasipolynomial or quasi-rational function with nonnegative integer exponents in their terms).

We say that the quasipolynomial (1) is *t-sparse* if at most t of the c_I are nonzero. If a quasi-rational function f can be written as a quotient of a numerator that is t_1-sparse and a denominator that is t_2-sparse, then we say that f is *(t_1, t_2)-sparse*. For example, $(X^m - 1)/(X - 1) = X^{m-1} + \dots + 1$ is $(2, 2)$-sparse and also $(m, 1)$-sparse. If f is (t_1, t_2)-sparse but not $(t_1 - 1, t_2)$- or $(t_1, t_2 - 1)$-sparse, we say that f is *minimally (t_1, t_2)-sparse*. Note that the above example is both minimally (2.2)-sparse and minimally $(m, 1)$-sparse. We say that a representation $f = p/q$ is a minimal (t_1, t_2)-sparse representation if f is minimally (t_1, t_2)-sparse and p is t_1-sparse and q is t_2-sparse.

We will need a zero test for (t_1, t_2)-sparse rational functions. This is similar to the well-known zero test for t-sparse polynomials (cf., [1], [9], [11]). We assume that we are given a black box for an n-variable rational function f with integer coefficients in which we can put points with rational coefficients. The output of the black box is either the value of the function at this point or some special sign (e.g., "∞") if the denominator of the irreducible representation of the function vanishes at this point. (A representation $f = g/h$, $g, h \in \mathbb{C}[X_1, \dots, X_n]$, is irreducible if g and h are relatively prime.)

LEMMA 1. *Let f be a (t_1, t_2)-sparse rational function of n variables, let $p_1, \dots, p_n$ be n distinct primes, and let $P^j = (p_i^j, \dots, p_n^j) 1 \le j \le t_1 + t_2 - 1$. Then f is not identically zero if and only if the black box outputs a number different from* 0 *and* ∞ *at one of the points p^j.*

Proof. If $M_1, \dots, M_t$ are distinct positive numbers, then any $t \times t$ subdeterminant of the $r \times t$ matrix $(M_s^j)_{1 \le s \le t, 1 \le j \le r}$ is nonsingular (cf. [5]). Since the black box gives output based on an irreducible representation of f, we see that any zero of the denominator of such a representation is zero of the denominator of a (t_1, t_2)-sparse representation of f. Using the remark about the matrix (M_s^j) above we see that the denominator can vanish at (at most) $t_2 - 1$ of these points. A similar argument applies to the numerator. Therefore, the (t_1, t_2)-sparse function f is not identically zero if and only if the black box outputs a number different from 0 and ∞ at one of these points P^j.

We note that Lemma 1 is not true for quasi-rational functions. For example, let $p = 2$ and $f(X) = 1 - X^{\frac{2\pi\sqrt{-1}}{\log 2}}$. We then have that $f(2^i) = 0$ for all i. If one restricts oneself to real quasi-rational functions, then Lemma 1 is also not true for $n \geq 2$. To see this, let $f(X_1, X_2) = X_1^{\log_2 5} - X_2^{\log_3 5}$ and $p_1 = 2$, $p_2 = 3$. However, we do have a zero test for univariate real quasi-rational functions. We need such a test, which we state in the following lemma, for real quasipolynomials only.

LEMMA 2. *Let p be a positive real number and let $f \in \mathbb{R}\langle X\rangle$ be t-sparse. If $f(p^i) = 0$ for $i = 0, \ldots, t-1$, then $f \equiv 0$.*

Proof. Let $f = \sum_{i=1}^{t} a_i X^{\alpha_i}$, where $\alpha_i \neq \alpha_j$ for $i \neq j$. Since $f(p^i) = 0$ for $i = 0, \ldots, t-1$,

$$\begin{bmatrix} 1 & \cdots & 1 \\ p^{\alpha_1} & \cdots & p^{\alpha_t} \\ \vdots & \vdots & \vdots \\ (p^{\alpha_1})^{t-1} & & (p^{\alpha_t})^{t-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_t \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Since the α_i are real, $p^{\alpha_i} \neq p^{\alpha_j}$ if $i \neq j$. Therefore, the above $t \times t$ matrix is nonsingular and so $a_1 = \cdots = a_t = 0$.

If f is a quasi-rational function, we call a representation $f = g/h$, $g, h \in \mathbb{C}\langle X_1, \ldots, X_n\rangle$ *normalized* if g or h contains the constant term 1. For an arbitrary representation $f = \tilde{g}/\tilde{h}$, there are a finite number of monomials M such that $(\tilde{g}/M)/(\tilde{h}/M)$ is normalized.

LEMMA 3. (a) *Assume that $p/q = \bar{p}/\bar{q}$ are normalized representations of a multivariate quasi-rational function and assume that p/q is a minimal (t_1, t_2)-sparse representation. Then the $\mathbb{Z}$-module generated by the exponent vectors of p and q is a submodule of the $\mathbb{Z}$-module generated by the exponent vectors of $\bar{p}$ and $\bar{q}$.*

(b) *There exist at most $(t_1 + t_2)^{O(t_1+t_2)}$ minimal (t_1, t_2)-sparse normalized representations. Furthermore, for given exponent vectors, the coefficients in the corresponding minimal representations are unique.*

(c) *Assume the same conventions as in* (a). *Then*

$$\max\{|\deg(p)|, |\deg(q)|\} \leq 2(t_1 + t_2) \max\{|\deg(\bar{p})|, |\deg(\bar{q})|\}.$$

Proof. Let $I_1, \ldots, I_{t_1}$ be the exponent vectors of p; $J_1, \ldots, J_{t_2}$ be the exponent vectors of q; and let $\{\bar{I}_i\}$ (respectively, $\{\bar{J}_j\}$) be the exponent vectors of $\bar{p}$ (respectively, $\bar{q}$). We define a weighted directed graph $\mathcal{G}$ in the following way: The vertices of $\mathcal{G}$ correspond to the $t_1 + t_2$ exponents of p/q. We join I_i and J_i if $I_i + \bar{J}_{j_1} = J_j + \bar{I}_{i_1}$ for some i_1, j_1 and assign the weight $\bar{I}_{i_1} - \bar{J}_{j_1}$ to the edge (I_i, J_j). We join I_i and I_{i_1} if $I_i + \bar{J}_j = I_{i_1} + \bar{J}_{j_1}$ for some $j \neq j_1$ and assign weight $\bar{J}_{j_1} - \bar{J}_j$ to the edge (I_i, I_{i_1}). Finally, we join J_j and J_{j_1} if $J_j + \bar{I}_i = J_{j_1} + \bar{I}_{i_1}$ for some $i \neq i_1$ and assign weight $\bar{I}_{i_1} - \bar{I}_i$ to the edge (J_j, J_{j_1}).

We claim that $\mathcal{G}$ is connected when viewed as an undirected graph. If not, let $\mathcal{G}_o$ be the connected component that contains the exponent vector $(0, \ldots, 0)$. One sees that the representation p_o/q_o obtained from p/q by deleting all terms with exponent vectors not belonging to this connected component equals $\bar{p}/\bar{q}$. This contradicts the minimality of p/q and proves the claim.

To prove (a) and (c), consider a spanning tree $\mathcal{T}$ of $\mathcal{G}$ and let $(0, \dots, 0)$ be the root of $\mathcal{T}$. Any exponent vector I_i (respectively, J_i) equals the sum of the weights along the unique path connecting I_i, (respectively, J_i) with the root and so lies in the module generated by the $\bar{I}_i$ and $\bar{J}_i$.

To prove (b), note that the spanning tree above uniquely determines the set of exponent vectors that can occur in p/q. Therefore, the number of exponent vectors in the numerator and denominator is at most the product of the number of such weighted trees and $\binom{t_1+t_2}{t_1}$ (the latter value being the number of choices of exponents for the numerator and denominator). The number of rooted trees with $(t_1 + t_2)$ vertices is at most $(t_1 + t_2)^{0(t_1+t_2)}$. For a fixed tree, the number of ways to assign weights of the above form from a fixed set $\{\bar{I}_i\}_{i=1}^{t_1} \cup \{\bar{J}_j\}_{j=1}^{t_2}$ can be bounded by $(t_1 + t_2)^{0(t_1+t_2)}$. Thus the number of exponent vectors can also be bounded by $(t_1 + t_2)^{0(t_1+t_2)}$.

We now prove the last statement of (b). Assume that $p_o/q_o = p/q$ are two different minimal (t_1, t_2)-sparse representations with the same exponent vectors in the corresponding numerators and denominators. For suitable $c \in \mathbb{C}$, $(p_o - cp)/(q_o - cq) = p/q$ is a representation that is either $(t_1 - 1, t_2)$- or $(t_1, t_2 - 1)$-sparse, contradicting the minimality of (t_1, t_2). This completes the proof of Lemma 3. □

We note that we do not know whether the above bound on the number of distinct minimal (t_1, t_2)-sparse representations is sharp. We have the following immediate consequence of Lemma 3(a).

COROLLARY 4. *Any normalized minimal (t_1, t_2)-sparse quasi-rational representation of a rational function has exponents that are integers.*

2. Linear operators.

2. Linear operators. In the following sections it will be useful to consider the actions of certain linear operators on fields of quasi-rational functions.

DEFINITION. (a) Let $p_1, \dots, p_n$ be distinct prime numbers, and let

$$D_n : \mathbb{C}\langle\langle X_1, \dots, X_n\rangle\rangle \to \mathbb{C}\langle\langle X_1, \dots, X_n\rangle\rangle$$

be the $\mathbb{C}$-linear operator defined by $D_n(X_i^{\alpha}) = p_i^{\alpha} X_i^{\alpha}$, where the number p_i^{α} is defined to be $e^{\alpha \log p_i}$ for some fixed branch of the logarithm. When $n = 1$ we will write $\mathbb{C}\langle\langle X\rangle\rangle$ instead of $\mathbb{C}\langle\langle X_1\rangle\rangle$ and D instead of D_1.

(b) Let $\mathcal{D} : \mathbb{C}\langle\langle X\rangle\rangle \to \mathbb{C}\langle\langle X\rangle\rangle$ be the $\mathbb{C}$-linear operator defined by

$$\mathcal{D}(X^{\alpha}) = X\frac{d}{dX}(X^{\alpha}) = \alpha X^{\alpha}.$$

We first note that given a black box for f, one can immediately construct a black box for $D_n f$. As we shall see below, this new black box will help us to interpolate f. D_n is used in a spirit similar to the prime power evaluation scheme of [1], [9], [11], and [12]. On the other hand, we know of no simple deterministic way of constructing a black box for $\mathcal{D}f$ given a black box for f, i.e., without first interpolating f. In this paper, our main use of $\mathcal{D}$ will be to prove that "faithful" mappings of multivariate (t_1, t_2)-sparse rational functions to univariate (t_1, t_2)-sparse rational functions are abundant. Note also that D_n is a homomorphism, i.e., $D_n(fg) = D_n(f)D_n(g)$ while $\mathcal{D}$ is a derivation, i.e., $\mathcal{D}(f)g + f\mathcal{D}(g)$. This difference will force us to deal with these operators separately. We begin by studying D_n.

LEMMA 5. (a) *Let $f \in \mathbb{C}(X_1, \dots, X_n)$, and assume that $D_n(f) = f$. Then $f \in \mathbb{C}$.*

(b) *Let $f \in \mathbb{R}\langle\langle X\rangle\rangle$ and assume that $D(f\ ) = f$. Then $f \in \mathbb{R}$.*

Proof. (a) If $D_n(f) = f$, then

$$F(X_1, \dots, X_n) = f(p_1X_1, \dots, p_nX_n) = f(p_1^2X_1, \dots, p_n^2X_n) = \cdots.$$

Lemma 1 implies that $f(X_1, \ldots, X_n) = f(X_1 Y_1, \ldots, X_n Y_n)$ for new variables $Y_1, \ldots, Y_n$. If $f = g/h$, let $g = \sum_I a_I \mathbf{X}^I, h = \sum_j b_J \mathbf{X}^J$. Comparing coefficients of the corresponding monomials in $\mathbf{X}$ and $\mathbf{Y}$ we have that, after a suitable reordering, $I_1 = J_1, I_2 = J_2, \ldots$ and $a_I b_J = a_J b_I$ for all I, J. Therefore, $f \in \mathbb{C}$.

(b) The proof is the same as in (a) using Lemma 2 instead of Lemma 1.

Note that Lemma 5(a) is not true for $f \in \mathbb{R}\langle\langle X_1, \ldots, X_n\rangle\rangle \subset \mathbb{C}\langle\langle X_1, \ldots, X_n\rangle\rangle, n \geq 2$. To see this, let $f = X_1^{\log_2 5} X_2^{-\log_3 5}$, $p_1 = 2$, $p_2 = 3$. Lemma 5(b) is not true for $f \in \mathbb{C}\langle\langle X\rangle\rangle$ since, for $p = 2$, $f = X^{\frac{2\pi\sqrt{-1}}{\log 2}}$ gives a counterexample.

LEMMA 6. (a) *If $y_1, \ldots, y_m \in \mathbb{C}(X_1, \ldots, X_n)$, then $y_1, \ldots, y_m$ are linearly dependent over $\mathbb{C}$ if and only if*

$$W_{D_n}(y_1, \ldots, y_m) = \det \begin{bmatrix} y_1 & \cdots & y_m \\ D_n y_1 & \cdots & D_n y_m \\ \vdots & \vdots & \vdots \\ D_n^{m-1} y_1 & \cdots & D_n^{m-1} y_m \end{bmatrix} = 0.$$

(b) *If $y_1, \ldots, y_m \in \mathbb{R}\langle\langle X\rangle\rangle$, then $y_1, \ldots, y_m$ are linearly dependent over $\mathbb{R}$ if and only if $W_{D_1}(y_1, \ldots, y_m) = 0$.*

Proof. (a) If $y_1, \ldots, y_m$ are linearly dependent over $\mathbb{C}$, then we clearly have $W_{D_n}(y_1, \ldots, y_m) = 0$. Now assume that $W_{D_n}(y_1, \ldots, y_m) = 0$. In this case there exist $f_1, \ldots, f_m \in \mathbb{C}(X_1, \ldots, X_n)$, not all zero, such that

$$f_1 y_1 + \cdots + f_m y_m = f_1 D_n y_1 + \cdots + f_m D_n y_m = \cdots = f_1 D_n^{m-1} y_1 + \cdots + f_m D_n^{m-1} y_m = 0.$$

We may assume $f_1 = 1$. Applying D_n to each of these equations, we have

$$D_n^i y_1 + D_n f_2 D^i y_2 + \cdots + D_n f_n D_n^i y_m = 0$$

for $i = 1, \ldots, n$. This implies that

$$(f_2 - D_n f_2) D_n^i y_2 + \cdots + (f_m - D_n f_m) D_n^i y_m = 0$$

for $i = 1, \ldots, n-1$. Either $f_i - D_n f_i = 0$ for $i = 2, \ldots, m$, in which case we are done by Lemma 5, or by induction there exist $\alpha_2, \ldots, \alpha_m \in \mathbb{C}$, not all zero, such that $\alpha_2 D_n y_2 + \cdots + \alpha_m D_n y_m = 0$. Therefore, $D_n(\alpha_2 y_2 + \cdots + \alpha_m y_m) = 0$, so $\alpha_2 y + \ldots + \alpha_m y_m = 0$. The proof of part (b) is similar and omitted.

Lemma 6 immediately implies the following criterion for a real quasi-rational function to be (t_1, t_2)-sparse.

LEMMA 7. (a) *Let $f \in \mathbb{C}(X_1, \ldots, X_n)$. f is (t_1, t_2)-sparse if and only if there exist $I_1, \ldots, I_{t_1}, J_1, \ldots, J_{t_2} \in \mathbb{Z}^n$, $I_i \neq I_j$, $J_i \neq J_j$ for $i \neq j$ such that*

$$W_{D_n}(\mathbf{X}^{I_1}, \ldots, \mathbf{X}^{I_{t_1}}, \mathbf{X}^{J_1} f, \ldots, \mathbf{X}^{J_{t_2}} f) = 0.$$

(b) *Let $f \in \mathbb{R}\langle\langle X\rangle\rangle$. f is (t_1, t_2)-sparse if and only if there exist $\alpha_1, \ldots, \alpha_{t_1}, \beta_1, \ldots, \beta_{t_2} \in \mathbb{R}$, $\alpha_i \neq \alpha_j$, $\beta_i \neq \beta_j$ for $i \neq j$ such that $W_D(X^{\alpha_1}, \ldots, X^{\alpha_{t_1}}, X^{\beta_1} f, \ldots, X^{\beta_{t_2}} f) = 0$.*

Proof. (a) f is (t_1, t_2)-sparse if and only if there exist $I_1, \ldots, I_{t_1}, J_1, \ldots, J_{t_2} \in \mathbb{Z}^n$, $I_i \neq I_j$, $J_i \neq J_j$ for $i \neq j$ and $a_1, \ldots, a_{t_1}, b_1, \ldots, b_{t_2} \in \mathbb{C}$, not all zero, such that $\sum_{i=1}^{t_1} a_i \mathbf{X}^{I_i} + \sum_{j=1}^{t_2} b_j \mathbf{X}^{I_j} f = 0$. By Lemma 6 this happens if and only if

$$W_{D_n}(\mathbf{X}^{I_1}, \ldots, \mathbf{X}^{I_{t_1}}, \mathbf{X}^{J_1} f, \ldots, \mathbf{X}^{J_{t_2}} f) = 0.$$

The proof of (b) is similar.

We now consider the other linear operator $\mathcal{D}$ on $\mathbb{C}\langle\langle X\rangle\rangle$. We will need results similar to Lemmas 5 and 6.

LEMMA 8. *If $f \in \mathbb{C}\langle\langle X\rangle\rangle$ and $\mathcal{D}f = 0$, then $f \in \mathbb{C}$.*

Proof. First assume that $f = \sum_{i=1}^{t} a_i X^{\alpha_i} \in \mathbb{C}\langle X\rangle$. If $0 = \mathcal{D}f = \sum_{i=1}^{t} a_i \alpha_i X^{\alpha_i}$, then $t = 1$ and $a_1 = 0$, so $f \in \mathbb{C}$.

Now let $f \in \mathbb{C}\langle\langle X\rangle\rangle$. f is minimally (t_1, t_2)-sparse for some (t_1, t_2). Let $f = g/h$ be a minimal (t_1, t_2)-sparse normalized representation. If $\mathcal{D}h = 0$, then we have just shown that $h \in \mathbb{C}$. Since $\mathcal{D}f = ((\mathcal{D}g)h - g\mathcal{D}h)/h^2 = (\mathcal{D}g)/h$, so $\mathcal{D}g = 0$. Therefore, $g \in \mathbb{C}$ and so $f \in \mathbb{C}$. We will, therefore, now assume that $\mathcal{D}h \neq 0$ and derive a contradiction. Since $(\mathcal{D}g)h - g\mathcal{D}h = 0$, we have $g/h = \mathcal{D}g/\mathcal{D}h$. Since g/h is normalized, $\mathcal{D}g/\mathcal{D}h$ is a $(t_1 - 1, t_2)$- or a $(t_1, t_2 - 1)$-sparse representation of f, a contradiction.

LEMMA 9. *If $y_1, \dots, y_m \in \mathbb{C}\langle\langle X\rangle\rangle$, then $y_1, \dots, y_m$ are linearly dependent over $\mathbb{C}$ if and only if*

$$W_{\mathcal{D}}(y_1, \dots, y_m) = \det \begin{bmatrix} y_1 & \cdots & y_m \\ \mathcal{D}y_1 & \cdots & \mathcal{D}y_m \\ \vdots & \vdots & \vdots \\ \mathcal{D}^{m-1}y_1 & \cdots & \mathcal{D}^{m-1}y_m \end{bmatrix} = 0.$$

Proof. Lemma 8 implies that $\mathbb{C}\langle\langle X\rangle\rangle$ is a differential field with constant subfield equal to $\mathbb{C}$. The result now follows from ([18, Thm. 3.7]).

3. Univariate interpolation. Lemma 7 in §2 allows us to characterize (t_1, t_2)-sparse rational functions and is the basis of the following algorithm for finding the exponents of a sparse univariate rational function.

Assume we are given a black box to evaluate a univariate rational function $f \in \mathbb{Q}(X)$, and assume we are told that it is minimally (t_1, t_2)-sparse. (The general case, in which we are told only that it is (t_1, t_2)-sparse, is handled below.) Consider the expression

$$\begin{aligned} S(p^{\alpha_1}, \dots, p^{\alpha_{t_1}}, p^{\beta_1}, \dots, &\ p^{\beta_{t_2}},\ f(X), f(pX), \dots, f(p^{t_1+t_2-1}X)) \\ &= \frac{W_{\mathcal{D}}(X^{\alpha_1}, \dots, X^{\alpha_{t_1}}, X^{\beta_1} f, \dots, X^{\beta_{t_2}} f\ )}{X^{\alpha_1} \cdot \dots \cdot X^{\alpha_{t_1}} \cdot X^{\beta_1} \cdot \dots \cdot X^{\beta_{t_2}}}. \end{aligned}$$

Note that S is a polynomial in the indicated terms with integer coefficients. Replacing $p^{\alpha_1}, \dots, p^{\alpha_{t_1}}, p^{\beta_1}, \dots, p^{\beta_{t_2}}$ with new variables $Y_1, \dots, Y_{t_1+t_2}$, we get a polynomial $S(Y_1, \dots, Y_{t_1+t_2}, f(X), f(pX), \dots, f(p^{t_1+t_2-1}X))$ with at most $(t_1 + t_2)^{t_1+t_2}$ terms in the variables $Y_1, \dots, Y_{t_1+t_2}$ and multilinear in the black boxes $f(X), f(pX), \dots, f(p^{t_1+t_2-1}X)$. Since we are looking for the exponents of a normalized minimal (t_1, t_2)-sparse representation of f, we may assume $Y_1 = 1$. By Lemma 7(b) $(0, \alpha_2, \dots, \alpha_{t_1}, \beta_1, \dots, \beta_{t_2}) \in \mathbb{R}^{(t_1+t_2)}$ will be a vector of such exponents if and only if

$$(2) \qquad S(1, p^{\alpha_2}, \dots, p^{\alpha_{t_1}},\ p^{\beta_1}, \dots, p^{\beta_{t_2}},\ f(X), f(pX), \dots) = 0$$

$$(3) \qquad 0 \neq \alpha_i \neq \alpha_j, \qquad \beta_i \neq \beta_j \quad \text{for} \quad i \neq j.$$

Observe that S as a rational function from $\mathbb{R}(X)$ is $((t_1 + t_2)^{2(t_1+t_2)}, t_2^{t_1+t_2})$-sparse, hence by Lemma 1, condition (2) is equivalent to the condition that S is either ∞ or 0 for $X =$

$p^i, i = 0, \dots, 2(t_1 + t_2 + 1)^{2(t_1+t_2)} - 1$. For at least $(t_1 + t_2 + 1)^{2(t_1+t_2)}$ of these points (being independent from $\alpha_2, \dots, \beta_{t_2}$), S will be zero. Using the black box for $f(X)$, we can determine a system T consisting of $(t_1+t_2+1)^{2(t_1+t_2)}$ equations in the unknowns $Y_2, \dots, Y_{t_1+t_2}$ of degree at most $(t_1 + t_2)$; of inequalities $1 \neq Y_i \neq Y_j \neq 1, 2 \leq i < j \leq t_1$, $Y_i \neq Y_j, t_1 < i < j \leq t_1 + t_2$; and of inequalities $Y_2 \geq 1, \dots, Y_{t_1+t_2} \geq 1$ that is equivalent to (2), (3) (for $Y_2 = p^{\alpha_2}, \dots, Y_{t_1+t_2} = p^{\beta_{t_2}}$). By Lemma 3(b), T has a finite number of solutions in $\mathbb{R}^{t_1+t_2-1}$. Note that Corollary 4 implies that these solutions are integers. We can apply the algorithm of [13], [14] (cf. also [1]) to this system and find these solutions with $\left((t_1 + t_2)^{(t_1+t_2)} \log d\right)^{O(1)}$ arithmetic operations and depth $((t_1 + t_2) \log d)^{O(1)}$, where d is the maximum of the exponents $\alpha_2, \dots, \beta_{t_2}$. Note that the algorithm of [13], [14] will yield a polynomial satisfied by these p-powers with $(t_1 + t_2)^{O(t_1+t_2)}$ arithmetic operations and $(t_1+t_2)^{O(1)}$ depth. As we noted in the introduction, the dependence on d of the final complexity is introduced when we find the roots of this polynomial. One can find these roots as in [23] or more simply by considering the powers of p that divide the coefficients. We remark that this algorithm also implies that there are at most $(t_1 + t_2)^{O(t_1+t_2)}$ solutions [cf. Lemma 3(b)] and that these solutions $p^{\alpha_2}, \dots, p^{\beta_{t_2}}$ are bounded by $p^d \leq \exp(M(t_1 + t_2)^{O(t_1+t_2)})$, where M is bound on the bitsize of the values yielded by the black box when we evaluate $f(p^{i+j})$ for $i = 0, \dots, t_1 + t_2 - 1$, $j = 0, \dots, 2(t_1 + t_2 + 1)^{2(t_1+t_2)} - 1$. Hence, the exponents $\alpha_2, \dots, \beta_{t_2}$ of the rational function f do not exceed $d \leq M(t_1 + t_2)^{O(t_1+t_2)}$. Notice that the algorithm can find the exponents $\alpha_2, \dots, \beta_{t_2}$ in $\left((t_1 + t_2)^{(t_1+t_2)} \log d\right)^{O(1)}$ arithmetic operations with the depth $((t_1 + t_2) \log d)^{O(1)}$.

We can find the coefficients by solving a system of linear equations gotten from

$$\left(\sum_{i=1}^{t_2} b_i X^{\beta_i}\right) f(X) = \sum_{i=1}^{t_1} a_i X^{\alpha_i}$$

by letting $X = p^j$, $j = 0, 1, \dots, t_1 + t_2 - 1$. Note that Lemma 3(b) implies that this system will have a unique solution. This can be found with $(t_1 + t_2)^{O(1)}$ arithmetic operations with depth $(\log(t_1 + t_2))^{O(1)}$, since to set up this system one has to compute powers p^{α_i}, p^{β_j}, which were computed above.

Turning to the general case, where we are told only that f is (t_1, t_2)-sparse, we proceed as follows: We consider all pairs (t_1', t_2') with $1 \leq t_1' \leq t_1, 1 \leq t_2' \leq t_2$ and use the above algorithm for these pairs. The first time that the above algorithm yields a nonempty set of solutions, we are guaranteed that, for this (t_1', t_2'), f has a minimal (t_1', t_2')-sparse representation and that the algorithm has yielded the exponents and the coefficients.

4. Multivariate interpolations. Let $f(X_1, \dots, X_n) \in \mathbb{Q}(X_1, \cdots, X_n)$ be a minimally (t_1, t_2)-sparse rational function given by a black box. We shall show in this section how the problem of finding the exponent vectors of f can be reduced to the univariate case. In particular, we shall show that the set of vectors $\boldsymbol{\nu} = (\nu_1, \dots, \nu_n) \in \mathbb{C}^n$ such that $f_{\boldsymbol{\nu}}(X) = f(X^{\nu_1}, \dots, X^{\nu_n})$ is not minimally (t_1, t_2)-sparse is a small set V. We will then show that if we find the exponents of $f_{\boldsymbol{\nu}}$ for sufficiently many $\boldsymbol{\nu} \notin V$, then we can recover the exponents appearing in f.

LEMMA 10. *Let $f(X_1, \dots, X_n)$ be a minimally (t_1, t_2)-sparse rational function and let $\nu_1, \dots, \nu_n \in \mathbb{C}$ be linearly independent over $\mathbb{Z}$. Then $f(X^{\nu_1}, \dots, X^{\nu_n})$ is minimally (t_1, t_2)-sparse.*

Proof. Let $\tilde{p}(X)/\tilde{q}(X)$ be a minimally $(\tilde{t}_1, \tilde{t}_2)$-sparse representation of $f(X^{\nu_1}, \dots, X^{\nu_n})$ with $\tilde{t}_1 \leq t_1, \tilde{t}_2 \leq t_2$. By Lemma 3(a), we may assume that $\tilde{p}, \tilde{q} \in \mathbb{C}[X^{\nu_1}, \dots, X^{\nu_n}]$. Since the map sending X^{ν_i} to X_i induces an isomorphism of $\mathbb{C}(X^{\nu_1}, \dots, X^{\nu_n})$ onto $\mathbb{C}(X_1, \dots, X_n)$, we get a $(\tilde{t}_1, \tilde{t}_2)$-sparse representation of $f(X_1, \dots, X_n)$. Therefore, $\tilde{t}_1 = t_1$, and $\tilde{t}_2 = t_2$.

LEMMA 11. *Let f be a minimally (t_1, t_2)-sparse rational function with integer coefficients. The set V of vectors $\boldsymbol{\nu} \in \mathbb{C}^n$ such that $f_{\boldsymbol{\nu}}$ is not minimally (t_1, t_2)-sparse lies in the union of at most $(t_1 + t_2)^{O((t_1+t_2)n)}$ hyperplanes determined by linear forms with integer coefficients.*

Proof. We will first show that V is defined by a set of polynomial equalities and inequalities with coefficients in $\mathbb{Q}$ (i.e., V is a $\mathbb{Q}$-constructible set). Let $V_1, \dots, V_n$ be variables. We shall write down conditions on $V_1, \dots, V_n$ so that $f(X^{V_1}, \dots, X^{V_n})$ is $(t_1 - 1, t_2)$-sparse. These conditions will be used to determine a set $\mathcal{W}^{(1)}$. Similar conditions can be derived for $f(X^{V_1}, \dots, X^{V_n})$ to be $(t_1, t_2 - 1)$-sparse and used to determine a set $\mathcal{W}^{(2)}$. Thus $\mathcal{W} = \mathcal{W}^{(1)} \cup \mathcal{W}^{(2)}$. Lemma 9 implies that $f(X^{V_1}, \dots, X^{V_n})$ is $(t_1 - 1, t_2)$-sparse if and only if there exist $\alpha_1, \dots, \alpha_{t_1-1}, \beta_1, \dots, \beta_{t_2} \in \mathbb{C}$ such that $\alpha_i \neq \alpha_j$, $\beta_i \neq \beta_j$ for $i \neq j$, and

$$\begin{aligned} & S_{\mathcal{D}}\left(\alpha_1, \dots, \alpha_{t_1-1}, \beta_1, \dots, \beta_{t_2}, f(X^{V_1}, \dots, X^{V_n}), \dots, \mathcal{D}^{t_1+t_2-2} f(X^{V_1}, \dots, X^{V_n})\right) \\ (4) \qquad & = \frac{W_{\mathcal{D}}(X^{\alpha_1}, \dots, X^{\alpha_{t_1}-1}, X^{\beta_1} f(X^{V_1}, \dots, X^{V_n}), \dots, X^{\beta_{t_2}} f(X^{V_1}, \dots, X^{V_n}))}{X^{\alpha_1} \cdot \dots \cdot X^{\alpha_{t_1-1}} \cdot X^{\beta_1} \cdot \dots \cdot X^{\beta_{t_2}}} \\ & = 0. \end{aligned}$$

When we clear the denominator of (4), we will get a linear function in expressions of the form $X^{\Sigma a_i V_i}$ with coefficients C_a, where $a = (a_1, \dots, a_n) \in \mathbb{Z}^n$, that are polynomials in $\alpha_1, \dots, \alpha_{t_1-1}, \beta_1, \dots, \beta_{t_2}, V_1, \dots, V_n$ with integer coefficients. Observe that there are at most $(t_1 + t_2)^{O(t_1+t_2)}$ distinct powers $X^{\Sigma a_i V_i}$ that can appear.

For any pair $\sum a_i V_i$, $\sum b_i V_i$ of distinct exponents, let $L_{a,b} = \sum (a_i - b_i) V_i$. Lemma 9 states that for any choice $(\nu_1, \dots, \nu_n) \in \mathbb{C}^n$ such that $L_{a,b}(\nu_1, \dots, \nu_n) \neq 0$, f is $(t_1 - 1, t_2)$-sparse if and only if there exist $\alpha_1, \dots, \alpha_{t_1-1}, \beta_1, \dots, \beta_{t_2} \in \mathbb{C}$ such that all the C_a considered above vanish. Let Φ be the formula, from the language of algebraically closed fields, with bound variables $\alpha_1, \dots, \alpha_{t_1-1}, \beta_1, \dots, \beta_{t_2}$ and free variables $V_1, \dots, V_n$, that express this latter statement. This formula contains at most $(t_1 + t_2)^{O(t_1+t_2)}$ polynomials, each of degree at most $(t_1 + t_2)^2$.

Applying the results of [6] (see also [4]), we can eliminate quantifiers and get a quantifier-free formula Ψ in variables $V_1, \dots, V_n$ equivalent to Φ. Furthermore, the polynomials occurring in Ψ have degrees at most $(t_1 + t_2)^{O((t_1+t_2)n)}$ and there are at most $(t_1 + t_2)^{O((t_1+t_2)n)}$ of these. This formula determines a constructible set $\mathcal{W}_0 \subset \mathbb{C}^n$. As it was shown previously, the symmetric difference $(\mathcal{W}^{(1)} \backslash \mathcal{W}_0) \cup (\mathcal{W}_0 \backslash \mathcal{W}^{(1)})$ lies in a union of all $(t_1 + t_2)^{O(t_1+t_2)}$ hyperplanes of the kind $L_{a,b}$ for the integer vectors a, b considered above. From Lemma 10, we know that for each point $(\nu_1, \dots, \nu_n) \in \mathcal{W}$ there exists a relation $\sum_{i=1}^n \gamma_i \nu_i = 0$ for suitable integers $\gamma_1, \dots, \gamma_n$ not all zero. From Lemma 12 of the appendix we know that each irreducible component of $\mathcal{W}_0$ (and also of $\mathcal{W}$) lies in a hyperplane. Therefore, $\mathcal{W}$ lies in the union of at most $(t_1 + t_2)^{O((t_1+t_2)n)}$ hyperplanes determined by linear forms with integer coefficients.

We now proceed to describe an algorithm to find p-powers of the exponents of a minimally (t_1, t_2)-sparse normalized rational function f.

For any $c > 0$ using the construction from ([11] or [12], Lem.), one can explicitly produce, for suitable $c_1 > 0$, $c_2 > 0$, $N = (t_1 + t_2)^{c_1(t_1+t_2)n}$ vectors $\boldsymbol{\nu}^{(i)} = (\nu_1^{(i)}, \dots, \nu_n^{(i)})$, $1 \leq i \leq N$, where the integers $1 \leq \nu_j^{(i)} \leq (t_1 + t_2)^{c_2(t_1+t_2)_n}$ such that for any family of $(t_1 + t_2)^{c(t_1+t_2)n}$ hyperplanes (containing the origin) at least n of these vectors lie in none of these hyperplanes and any n of these vectors are linearly independent. We take $c > 0$ such that the number of hyperplanes in Lemma 11 is at most $(t_1 + t_2)^{c(t_1+t_2)n}$ (so for the algorithm we have only to estimate explicitly constant c once and forever) and apply to this c the construction mentioned above. For each of the vectors $\boldsymbol{\nu}^{(i)}$ produced in this way, use the algorithm from §3 to find $t_1^{(i)} \leq t_1$, $t_2^{(i)} \leq t_2$ such that the rational function $f_{\boldsymbol{\nu}^{(i)}} \in \mathbb{Q}(X)$ has a minimal $(t_1^{(i)}, t_2^{(i)})$-sparse representation. By Lemma 11 and the construction of the $\boldsymbol{\nu}^{(i)}$, there exist at least n vectors among the $\boldsymbol{\nu}^{(i)}$ (without loss of generality we let them be $\boldsymbol{\nu}^{(1)}, \dots, \nu^{(n)}$) such that $f_{\boldsymbol{\nu}^{(i)}}$ is minimally (t_1, t_2)-sparse for all $1 \le i \le n$. Using the algorithm from §3 we find p-powers of the exponents of all normalized (t_1, t_2)-sparse representations of $f_{\boldsymbol{\nu}^{(i)}}$ for each $1 \le i \le n$ (recall that there are at most $(t_1 + t_2)^{O(t_1+t_2)}$ of these). For each $f_{\boldsymbol{\nu}^{(i)}}$, $1 \le i \le n$, pick out one set of such p-powers of the exponents $p^{\alpha_1^{(i)}}, \dots, p^{\alpha_{t_1}^{(i)}}, p^{\beta_1^{(i)}}, \dots, p^{\beta_{t_2}^{(i)}}$. For each i, $1 \le i \le n$, we also pick out two permutations $\pi^{(i)} \in \mathcal{S}_{t_1}$ and $\sigma^{(i)} \in \mathcal{S}_{t_2}$, where $\mathcal{S}_m$ is the permutation group on m elements. For every j_i, $1 \le j_1 \le t_1$, the algorithm solves the p-power form of a linear system

$$p^{\sum_{k=1}^{n} \nu_k^{(i)} Y_k^{(j_1)}} = p^{\alpha^{(i)}_{\pi^{(i)}(j_1)}} \tag{5}$$

$$1 \le i \le n$$

and for every j_2, $1 \le j_2 \le t_2$ a system

$$p^{\sum_{k=1}^{n} \nu_k^{(i)} Z_k^{(j_2)}} = p^{\beta^{(i)}_{\sigma^{(i)}(j_2)}} \tag{6}$$

$$1 \le i \le n.$$

Using [22] the algorithm produces the inverse matrix $(\mu_k^{(i)}/\mu)$, where $\mu_k^{(i)}$, $\mu \in \mathbb{Z}$, to $n \times n$ matrix $(\nu_k^{(i)})$, which is invertible because of the construction of the vectors $\nu^{(i)}$. Then

$$p^{\mu Y_k^{(j_1)}} = p^{\sum_{1 \le i \le n} \mu_k^{(i)} \alpha^{(i)}_{\pi^{(i)}(j_1)}}$$

and the algorithm computes the right side of this equality. The algorithm also computes $p^{\mu Z_k^{(j_2)}}$. The vectors $\mathbf{Y}^{(1)} = (Y_1^{(1)}, \dots, Y_n^{(1)}), \dots, \mathbf{Y}^{(t_1)} = (Y_1^{(t_1)}, \dots, Y_n^{(t_1)})$ and $\mathbf{Z}^{(1)} = (Z_1^{(1)}, \dots, Z_n^{(1)}), \dots, \mathbf{Z}^{(t_2)} = (Z_1^{(t_2)}, \dots, Z_n^{(t_2)})$ are considered as candidates for being exponent vectors in the numerator and denominator of a (t_1, t_2)-sparse representation of f. The algorithm represents them by $p^{\mu Y_k^{(j_1)}}$, $p^{\mu Z_k^{(j_2)}}$. The algorithm tests whether $\mathbf{Y}^{(j)} \ne \mathbf{Y}^{(l)}$, $\mathbf{Z}^{(j)} \ne \mathbf{Z}^{(l)}$ for $j \ne 1$. Then the algorithm tests whether these candidates fit. For this aim consider the linear system

$$\textstyle\sum_{1 \le i \le t_1} \phi_i p_1^{\mu Y_1^{(i)} l} \cdots p_n^{\mu Y_n^{(i)} l} = \sum_{1 \le i \le t_2} \psi_i p_1^{\mu Z_1^{(i)} l} \cdots p_n^{\mu Z_n^{(i)} l} f(p_1^{\mu l}, \dots, p_n^{\mu l}), \tag{7}$$

$$1 \le l \le 2(t_1 + t_2)^2$$

in the unknown coefficients ϕ_i, ψ_i of the (t_1, t_2)-sparse representations of f currently being tested. (In (7) we skip the equations for which $f(p_1^{\mu l}, \dots, p_n^{\mu l}) = \infty$.) Lemma 1 implies that (7) is solvable if and only if exponent vectors $\mathbf{Y}^{(j)}$, $\mathbf{Z}^{(j)}$ fit. Here we apply Lemma 1 but probably not to rational functions, since the exponents $Y_k^{(i)}$, $Z_k^{(i)}$ could be rational; it is, however, still valid when the variables $X_i \to \bar{X}_i^{\mu l}$, $1 \le i \le n$ are replaced.) If (7) is solvable, then $Y_k^{(i)}$, $Z_k^{(i)}$ are integers because of Lemma 3(a). Moreover, it has a unique solution by Lemma 3(b). This completes the description of the algorithm for f as minimally (t_1, t_2)-sparse. To treat the case when we are told only that f is (t_1, t_2)-sparse, we proceed as in §3.

Now we proceed to the complexity bounds. Let us assume we are given the black box for a (t_1, t_2)-sparse rational function. The algorithm produces $(t_1 + t_2)^{O((t_1+t_2)n)}$ integer vectors $\boldsymbol{\nu}^{(i)}$

and, for each of these, applies the algorithm from §3 to the univariate rational function $f_{\nu^{(i)}}$. This part of the algorithm requires $\left((t_1+t_2)^{(t_1+t_2)n}\log d\right)^{O(1)}$ arithmetic operations with depth $((t_1+t_2)n\log d)^{O(1)}$. The algorithm then selects, for each i, $1\le i\le n$, some (t_1,t_2)-sparse representation of $f_{\nu^{(i)}}$ and also two permutations $\pi^{(i)}, \sigma^{(i)}$. This is again within the same bounds. The algorithm then solves $(t_1+t_2)^{O((t_1+t_2)n)}$ p-power forms of linear systems of type (5) and (6). To invert $n\times n$ matrix $(\nu_k^{(i)})$, $n^{O(1)}$ arithmetic operations are used with depth $\log^{O(1)} n$. Since $\mu_k^{(i)}, \mu \le (t_1+t_2)^{O((t_1+t_2)n^2)}$ computation of p^{μ}, $p^{\mu Y_k^{(j_1)}}$, $p^{\mu Z_k^{(j_2)}}$ can be done within the same complexity bounds. The same applies to solving system (7). If we are told only that f is (t_1,t_2)-sparse, the additional search required by the algorithm does not change the complexity.

We are also able to give some bounds on the degree d of a sparse representation. Assume that A is a bound for all the exponents $\alpha_j^{(i)}$, $\beta_j^{(i)}$ found for the univariate rational functions $f_{\nu^{(i)}}$ (such a bound can be found using the techniques of §3). We can then bound d by looking at p-power forms of the linear systems (5) and (6); in fact, $d\le A(t_1+t_2)^{O((t_1+t_2)n^2)}$. Thus, we can formulate the main result of the paper, which is given in the following theorem.

THEOREM. (a) *One can construct some (t_1,t_2)-sparse representation*

$$\sum_{1\le i\le t_1} a_i X_1^{j_1^{(i)}}\cdots X_n^{j_n^{(i)}} \Big/ \sum_{1\le i\le t_2} b_i X_1^{k_1^{(i)}}\cdots X_n^{k_n^{(i)}}$$

of (t_1,t_2)-sparse rational function f in $\left((t_1+t_2)^{(t_1+t_2)n}\log d\right)^{O(1)}$ arithmetic operations with the depth $((t_1+t_2)n\log d)^{O(1)}$.

(b) *The exponents $j_l^{(i)}, k_l^{(i)}$ do not exceed $d\le M(t_1+t_2)^{O((t_1+t_2)n^2)}$, where M is the bound on bitsizes of all the outputs of applications of a black box during the computation.*

Appendix. For the convenience of the reader, we give a short proof of the result about complex varieties that was needed in the proof of Lemma 11. This result is true for varieties over any algebraically closed field of characteristic 0, but the proof is more complex and depends on the Hilbert Irreducibility Theorem instead of on elementary topological notions.

LEMMA 12. *Let $\mathcal{W}$ be an irreducible constructible set in $\mathbb{C}^n$ (i.e., a constructible set whose Zariski closure is irreducible). Assume that for each $\nu=(\nu_1,\dots,\nu_n)\in\mathcal{W}$ there exist $\gamma_1,\dots,\gamma_n\in\mathbb{Z}$, not all zero, such that $\sum_{i=1}^n\gamma_i\nu_i=0$. Then there exist $\tilde\gamma_1,\dots,\tilde\gamma_n\in\mathbb{Z}$, not all zero, such that $\sum_{i=1}^n\tilde\gamma_i\nu_i=0$ for all $(\nu_i,\dots,\nu_n)\in\mathcal{W}$.*

Proof. If $\mathcal{W}$ has dimension 0, then it is a point and we are done. Therefore, assume $\dim\mathcal{W}>0$ and let $\overline{\mathcal{W}}$ be its Zariski closure. Since $\mathcal{W}$ is constructible, it can be written as a finite union of sets $\mathcal{W}_i$, each of which is Zariski open in its Zariski closure $\overline{\mathcal{W}_i}$. Since $\mathcal{W}$ is irreducible, we must have that for some i, $\overline{\mathcal{W}}=\overline{\mathcal{W}_i}$. Therefore $\mathcal{W}$ contains a set $\mathcal{D}$ that is Zariski open in $\overline{\mathcal{W}}$. Since the nonsingular points of $\overline{\mathcal{W}}$ are Zariski dense, there exists a point $\nu\in\mathcal{W}$ that is nonsingular in $\overline{\mathcal{W}}$ (i.e., a point where the Jacobian of a suitable system of defining equations has maximal rank). We select a sufficiently small ε such that $\mathcal{W}_\varepsilon=\mathcal{W}\cap\{\mathbf{x}\mid\|\mathbf{x}-\nu\|\le\varepsilon\}$ will be closed in the usual topology and contain an open subset of $\mathcal{W}$. For each $(\gamma_1,\dots,\gamma_n)\in\mathbb{Z}^n$, not all γ_i zero, let $H_{\gamma_1,\dots,\gamma_n}=\{(\nu_1,\dots,\nu_n)\in\mathcal{W}\mid\sum_{i=1}^n\gamma_i\nu_i=0\}$. Since $\mathcal{W}_\varepsilon$ is closed, the Baire Category Theorem ([24, p. 139]) implies that for some $(\tilde\gamma_1,\dots,\tilde\gamma_n)$, $H_{\tilde\gamma_1,\dots,\tilde\gamma_n}$ contains an open subset of $\mathcal{W}_\varepsilon$ (and so, of $\mathcal{W}$). Therefore, $\dim(H_{\tilde\gamma_1,\dots,\tilde\gamma_n}\cap\overline{\mathcal{W}})=\dim\overline{\mathcal{W}}$. Since $\overline{\mathcal{W}}$ is irreducible, we must have $\dim(H_{\tilde\gamma_1,\dots,\tilde\gamma_n}\cap\overline{\mathcal{W}})=\overline{\mathcal{W}}$ (cf. [26, p. 54]) so $\overline{\mathcal{W}}\subseteq H_{\tilde\gamma_1,\dots,\tilde\gamma_n}$.

Acknowledgment. We are indebted to Volker Strassen for motivating the problem and a number of stimulating discussions.

REFERENCES

[1] M. Ben-Or and P. A. Tiwari, *A deterministic algorithm for sparse multivariate polynomial interpolation*, Proceedings of the 20th Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 301–309.

[2] A. Borodin and P. A. Tiwari, *On the Decidability of Sparse Univariate Polynomial Interpolation*, Res. Report RC 14923, IBM T. J. Watson Research Center, New York, NY, 1989.

[3] A. L. Chistov, *An algorithm of polynomial complexity for factoring polynomials and finding the components of a variety in subexponential time*, J. Soviet Math., 34, 4 (1986), pp. 000–000.

[4] A. L. Chistov and D. Yu. Grigoriev, *Complexity of quantifier elimination in the first-order theory of algebraically closed fields*, in Lecture Notes in Computer Science 176, Springer-Verlag, New York, 1984, pp. 17–31.

[5] R. J. Evans and I. M. Isaacs, *Generalized Vandermonde determinants and roots of unity of prime order*, in Proc. Amer. Math. Soc., 58 (1976), pp. 000–000.

[6] N. Fitchas, A. Galligo, and J. Morgenstern, *Sequential and parallel complexity bounds for the quantifier elimination of algebraically closed fields*, J. Pure Appl. Algebra, 67 (1990), pp. 1–14.

[7] D. Yu. Grigoriev, *Factoring polynomials over a finite field and solving systems of algebraic equations*, J. Soviet Math., 34, 4 (1986), pp. 1762–1803.

[8] ———, *Complexity of deciding Tarski algebra*, J. Symbol. Comput. 5 (1988), pp. 65–108.

[9] D. Yu. Grigoriev and M. Karpinski, *The matching problem for bipartite graphs with polynomially bounded permanents is in NC*, in Proceedings of the 28th Annual Symposium on the Foundations of Computer Science, Institute for Electrical and Electronics Engineers, New York, 1987, pp. 166–172.

[10] D. Yu. Grigoriev, M. Karpinski, and M. Singer, *Interpolation of sparse rational functions without knowing bounds on exponents*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, 1990, pp. 840–847.

[11] ———, *Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields*, SIAM J. Comput., 19 (1990), pp. 1059–1063.

[12] ———, *The interpolation problem for k-sparse sums of eigenfunctions of operators*, Adv. in Appl. Math., 12 (1991), pp. 76–81.

[13] D. Yu. Grigoriev and N. N. Vorobjov, *Solving systems of polynomial inequalities in subexponential time*, J. Symbol. Comput., 5 (1988), pp. 37–64.

[14] J. Heintz, M.-F. Roy, and P. Solerno, *Complexité du principe de Tarski–Seidenberg*, C.R.A.S. Paris, t. 309 (1989), pp. 825–830.

[15] E. Kaltofen, *Uniform Closure Properties of P-computable Functions*, Proceedings of the 18th Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 330–337.

[16] E. Kaltofen and B. Trager, *Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators*, in Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science, Institute for Electrical and Electronics Engineers, New York, 1988, pp. 296–305.

[17] E. Kaltofen and L. Yagati, *Improved Sparse Multivariate Polynomial Interpolation*, Report 88-17, Dept. of Computer Science, Rensselaer Polytechnic Institute, New York, 1988.

[18] I. Kaplanski, *An Introduction to Differential Algebra*, Hermann, Paris, 1957.

[19] M. Karpinski, *Boolean circuit complexity of algebraic interpolation problems*, In Proc. CSL'88, Lecture Notes in Computer Science 385, Springer-Verlag, New York, 1989, pp. 138–147.

[20] M. Karpinski and F. Meyer auf der Heide, *On the complexity of genuine polynomial computation*, in Proc. 15th MFCS, Lecture Notes in Computer Science 452, Springer-Verlag, 1990, pp. 362-368.

[21] M. Karpinski and T. Werther, *VC Dimension and Learnability of Sparse Polynomials and Rational Functions*, SIAM J. Comput., 22 (1993), pp. 1276–1285.

[22] K. Mulmuley, *A fast parallel algorithm to compute the rank of a matrix over an arbitrary field*, in Proceedings of the 18th Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 338–339.

[23] V. Pan and J. Reif, *Some polynomial and Toeplitz matrix computations*, in Proceedings of the 28th Annual IEEE Symposium on the Foundations of Computer Science, Institute for Electrical and Electronics Engineers, New York, 1987, pp. 173–184.

[24] H. L. Royden, *Real Analysis*, 2nd ed., MacMillan, New York, 1971.

[25] V. Strassen, *Vermeidung von Divisionen*, J. Reine und Angewandte Math., 65 (1973), pp. 182–202.

[26] I. Shafarevich, *Basic Algebraic Geometry*, Springer-Verlag, New York, 1977.

[27] R. E. Zippel, *Probabilistic algorithms for sparse polynomials*, in Lecture Notes in Computer Science 72, Springer-Verlag, New York, 1979, pp. 216–226.

[28] ———, *Interpolation polynomials from their values*, J. Symbol. Comput., 9 (1990), pp. 375–403.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 12–23, February 1994

TIGHT UPPER AND LOWER BOUNDS ON THE PATH LENGTH OF BINARY TREES*

ALFREDO DE SANTIS† AND GIUSEPPE PERSIANO‡

Abstract. The *external path length* of a tree T is the sum of the lengths of the paths from the root to each external node. The *maximal path length difference*, Δ, is the difference between the lengths of the longest and shortest such paths.

Tight lower and upper bounds are proved on the external path length of binary trees with N external nodes and maximal path length difference Δ is prescribed.

In particular, an upper bound is given that, for each value of Δ, can be exactly achieved for infinitely many values of N. This improves on the previously known upper bound that could only be achieved up to a factor proportional to N. An elementary proof of the known upper bound is also presented as a preliminary result.

Moreover, a lower bound is proved that can be exactly achieved for each value of N and $\Delta \leq N/2$.

Key words. binary search trees, path length

AMS subject classifications. 68P05, 68P10

1. Introduction. Binary trees constitute the most important and widely used data structure for the storage and retrieval of information. The cost of accessing information stored in a node is proportional to the distance of the node from the root. An important measure of efficiency, often considered with respect to a binary tree T, is its *external path length* $\text{EXTERNAL}(T)$, which is the sum of the distances of the external nodes from the root as this is related to the average cost of an unsuccessful search in the tree T. Moreover, the external path length of T is related to the internal path length $\text{INTERNAL}(T)$, which is the sum of the distances of the internal nodes from the root and corresponds to the average cost of a successful search in the tree T.

It is well known that the external path length of a binary tree with N external nodes is $\Theta(N \lg N)$ in the best case but can be as bad as $\Theta(N^2)$ in the worst case. The large gap between the best and the worst cases motivates the study of this important quantity when some additional information about the tree is available. Nievergelt and Wong [7] proposed an upper bound for the external path length of a tree T in terms of the number of external nodes and the maximum weight balance of all its subtrees. More recently, Klein and Wood [5] obtained an upper bound that requires much less information about the tree. Namely, they derived an upper bound on the external path length of a T in terms of N, the number of external nodes, and Δ, the maximum of the differences of the lengths of the paths from the root to each external node. In the case $\Delta \leq \sqrt{N}$, they also constructed trees whose external path length is within $O(N)$ from their upper bound.

In this paper we continue and extend the work of [5]. We present an upper bound on the external path length of binary trees with N nodes and maximum path length difference Δ and construct an infinite class of trees whose external path length matches our bound exactly. Moreover, we present a lower bound on the external path length of these trees. For the case $\Delta \leq N/2$, there exist trees whose external path length matches exactly our bound.

*Received by the editors April 4, 1991; accepted for publication (in revised form) June 2, 1992.

†Dipartimento di Informatica ed Applicazioni, Università di Salerno, 84081 Baronissi (Salerno), Italy. Part of this work was done while the author was visiting IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York 10598. This author's research was partially supported by the Italian Ministry of University and Scientific Research (MURST) and National Research Council (CNR).

‡Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138. Part of this work was done while the author was visiting AT&T Bell Laboratories, Murray Hill, New Jersey 07947. This author's research was partially supported by National Science Foundation grant NSF-CCR-90-07677.

1.1. Organization of the paper and summary of the results. In this paper we study the path length of (N, Δ)-trees, which are extended binary trees with N external nodes and maximum path difference Δ.

In §2 we set up our notation and recall some elementary facts about binary trees.

In §3, we present our upper bounds. As a preliminary result, in §3.1 we give a simple derivation of the upper bound presented in [5]. In §3.2 we improve on this result by giving a better bound that, for *each* Δ, can be exactly achieved for infinitely many N.

In §4 we present our lower bound. We exactly compute, for each N and $\Delta \leq N/2$, the external path length of the tree with shortest external path length. Our lower bound is the first lower bound to improve on the banal lower bound obtained by considering the binary tree with external nodes on two consecutive levels, which has the shortest external path length among the trees with the same number of external nodes.

2. Background and notation.

2. Background and notation. In this section we set up our notation and recall some elementary facts about binary trees.

We denote the set of the natural numbers by $\mathcal{N}$, the set of the positive natural numbers by $\mathcal{N}^+$, and the set of the positive real numbers by $\mathcal{R}^+$. The writing $\lceil x \rceil$ denotes the least integer greater than or equal to x. Throughout this paper, the writing $\lg x$ denotes the logarithm to base 2 of x.

All the trees considered in this paper are *extended binary trees*. An *extended binary tree* is obtained from a binary tree by adding special nodes, the *external nodes*, so that every node in the binary tree has exactly two children. The originary nodes of the binary tree are also called internal nodes.

We say that an external node e is at level l of a tree T if the length of the (unique) path from the root to e is l. Let T have N external nodes at levels $l_1, \ldots, l_N$. Then, the *external path length* of T, $\textsc{External}(T)$, is defined as

$$\textsc{External}(T) = \sum_{i=1}^{N} l_i.$$

The internal path length of a tree $\textsc{Internal}(T)$ is defined as the sum of the levels of the *internal* nodes.

The *maximum path length difference* of T, $\textsc{Fringe}(T)$, is defined as

$$\textsc{Fringe}(T) = \max_{1 \leq i \leq N} l_i - \min_{1 \leq j \leq N} l_j.$$

Also, a level l (other than the last one) is said to be *dense* if it has more than one external node, while the last level is dense if it has more than two external nodes. An (N, Δ)-tree is a binary tree with N external nodes and maximum path length difference Δ.

For $2 \leq \Delta \leq N - 2$ define $epl(N, \Delta)$ and $\textsc{Epl}(N, \Delta)$ as

$$epl(N, \Delta) = \min_{T \text{ is a } (N, \Delta)\text{-tree}} \textsc{External}(T)$$

and

$$\textsc{Epl}(N, \Delta) = \max_{T \text{ is a } (N, \Delta)\text{-tree}} \textsc{External}(T).$$

In this paper we give bounds on $epl(N, \Delta)$ and $\textsc{Epl}(N, \Delta)$. We restrict our attention to the cases when $2 \leq \Delta \leq N - 2$ since if $\Delta > N - 2$ then there exists no (N, Δ)-tree and the cases $\Delta = 0, 1$ can be completely characterized (see §4).

The following is a simple and well-known result about extended binary trees (see, e.g., [4]) that will be extensively used in the sequel.

FACT 2.1 (Kraft equality). *Let $l_1, \dots, l_N$ be a sequence of positive integers. Then there exists an extended binary tree with N external nodes at levels $l_1, \dots, l_N$ if and only if $\sum_{i=1}^{N} 2^{-l_i} = 1$.*

We shall say that a (N, Δ)-tree T has *configuration* $(L; a_0, a_1, \dots, a_{\Delta})$ if

1. $a_0, a_\Delta \geq 1$;
2. T has a_i external nodes at level $L + i, 0 \leq i \leq \Delta$; and
3. $\sum_{i=0}^{\Delta} a_i = N$.

Two (N, Δ)-trees with the same configuration are said *isomorphic*, as for our purposes they are the "same" tree. In what follows, we will omit L from a configuration when it is either clear from the context or immaterial.

3. Upper bounds.

In this section we present upper bounds on $\text{EPL}(N, \Delta)$. The problem of computing $\text{EPL}(N, \Delta)$ can be stated as follows:

Find the maximum of $\sum_{i=1}^{N} l_i$ subject to

1. $l_i \in \mathcal{N}^+$ for $i = 1, \dots, N$;
2. $\sum_{i=1}^{N} 2^{-l_i} = 1$; and
3. $\max_i l_i - \min_j l_j = \Delta$.

Because of the constraints $l_i \in \mathcal{N}^+$, finding the exact maximum does not appear to be an easy task.

3.1. A simple proof of a known upper bound.

Recently, Klein and Wood [5] proved the following upper bound on $\text{EPL}(N, \Delta)$.

BOUND 1.

$$\text{EPL}(N, \Delta) \leq N(\lg N + \Delta - \lg \Delta - \Psi(\Delta))$$

where

$$\Psi(\Delta) = \lg e - \lg \lg e - \frac{\Delta}{2^{\Delta} - 1} - \lg\left(1 - \frac{1}{2^{\Delta}}\right) \geq 0.6622,$$

and e is the base of the natural logarithm.

In [5], the bound above is derived by first proving that the external path length is related to the ratio of the geometric and the harmonic means of certain integers and then applying a theorem by Specht [8]. In this section, we re-derive Bound 1 using only elementary calculus.

We obtain an upper bound on $\text{EPL}(N, \Delta)$ by allowing the l_i's to range over the real numbers.

To this end, for $2 \leq \Delta \leq N - 2$, we define $M(N, \Delta, x)$ as the maximum of $\sum_{i=1}^{N} l_i$ over the sequences $l_1, \dots, l_N$ satisfying the following constraints

1. $l_i \in \mathcal{R}^+$, for $i = 1, 2, \dots, N$;
2. $\sum_{i=1}^{N} 2^{-l_i} = 1$; and
3. $x \leq l_i \leq x + \Delta$, for $i = 1, 2, \dots, N$.

A triplet (N, Δ, x) for which there exists a sequence $l_1, \dots, l_{N}$ that satisfies the three constraints above is called *admissible*.

It is clear that an upper bound on the external path length of binary trees is given by

$$\text{EPL}(N, \Delta) \leq \max\{M(N, \Delta, x) | (N, \Delta, x) \text{ is admissible and } x \in \mathcal{R}^+\}. \tag{1}$$

The following lemma gives a useful characterization of the maximum $M(N, \Delta, x)$.

LEMMA 3.1. *Let (N, Δ, x) be an admissible triplet. The maximum $M(N, \Delta, x)$ is achieved by a sequence $l_1, l_2, \ldots, l_n$ where all the l_i's, but at most one, are equal either to x or $x + \Delta$.*

Proof. Assume, by way of contradiction, that the maximum is reached by $l'_1, l'_2, \ldots, l'_N$ such that $x < l'_1 \le l'_2 < x + \Delta$. Fix $\epsilon > 0$, and define the sequence $l''_1, l''_2, \ldots, l''_N$ as

$$l''_i = \begin{cases} l'_1 - \delta & \text{if } i = 1, \\ l'_2 + \epsilon & \text{if } i = 2, \\ l'_i & \text{otherwise,} \end{cases}$$

where δ is chosen so that the Kraft equality holds; that is, $2^{-l''_1} + 2^{-l''_2} = 2^{-l'_1} + 2^{-l'_2}$ and thus $\delta = \lg(1 + 2^{l'_1 - l'_2}(1 - 2^{-\epsilon}))$. Notice that for sufficiently small values of ϵ (i.e., $\epsilon < \min\{x + \Delta - l'_2, l'_1 - x\}$), l''_1 and l''_2 are both in $[x, x + \Delta]$. Since the function $x - \lg(1 + 2^{l'_2 - l'_1}(1 - 2^{-x}))$ is increasing in x and is equal to 0 for $x = 0$, it is positive for $x > 0$, and thus $\epsilon > \delta$ holds. Therefore, if $\epsilon < \min\{x + \Delta - l'_2, l'_1 - x\}$, the sequence $l''_1, \ldots, l''_N$ satisfies the three constraints in the definition of $M(N, \Delta, x)$ and $\sum_{i=1}^{N} l''_i > \sum_{i=1}^{N} l'_i$, contradicting the maximality of $l'_1, \ldots, l'_N$. □

The following lemma provides an upper bound on the maximum $M(N, \Delta, x)$.

LEMMA 3.2. *Let (N, Δ, x) be an admissible triplet. Then*

$$M(N, \Delta, x) \le Nx + \Delta \frac{N - 2^x}{1 - 2^{-\Delta}}.$$

Proof. By Lemma 3.1, the maximum is achieved when $l_1 = \cdots = l_m = x + \Delta$, $l_{m+1} = x + y$ and $l_{m+2} = \cdots = l_N = x$, for some $1 \le m \le N$ and $0 \le y < \Delta$. By the Kraft equality, we get

$$\frac{m}{2^{x+\Delta}} + \frac{1}{2^{x+y}} + \frac{N - m - 1}{2^x} = 1,$$

from which we see that m is uniquely specified by N, Δ, y, and x. The maximum $M(N, \Delta, x)$ is thus given by

$$\begin{aligned} M(N, \Delta, x) &= n = m(x + \Delta) + (x + y) + (N - m - 1)x \\ &= m\Delta + Nx + y \\ &= Nx + \Delta \frac{N - 2^x}{1 - 2^{-\Delta}} + y - \Delta \frac{1 - 2^{-y}}{1 - 2^{-\Delta}}. \end{aligned}$$

The function $f(y) = y - \Delta(1 - 2^{-y})/(1 - 2^{-\Delta})$ is a convex function of y with $f(0) = 0$ and $f(\Delta) = 0$. Thus $f(y) \le 0$ for $0 \le y \le \Delta$. Hence the lemma is proved. □

We are now ready to prove Bound 1.

THEOREM 3.3.

$$\mathrm{EPL}(N, \Delta) \le N(\lg N + \Delta - \lg \Delta - \Psi(\Delta))$$

where

$$\Psi(\Delta) = \lg e - \lg \lg e - \frac{\Delta}{2^\Delta - 1} - \lg\left(1 - \frac{1}{2^\Delta}\right),$$

and e is the base of the natural logarithm.

Proof. Combining Lemma 3.2 and (1) we get

$$EPL(N, \Delta) \le \max_{x \in \mathcal{R}^+} \left\{ Nx + \Delta \frac{N - 2^x}{1 - 2^{-\Delta}} \right\}. \tag{2}$$

The function to maximize is a concave function of x. It can easily be seen, using elementary calculus, that it reaches its maximum (for fixed values of N and Δ) at $x_{\max} = \lg N + \lg(1 - 2^{-\Delta}) + \lg\lg e - \lg \Delta$. Substituting this into (2) we get Bound 1. □

3.2. An improved upper bound achievable by infinitely many values. In the previous section all maximizations were carried over the real numbers. Now, we show that by restricting x to be an integer we get a better bound. The bound presented in this section improves on Bound 1 on two accounts: first, it is more accurate; second, for *each* value of Δ, it can be exactly achieved for infinitely many values of N.

BOUND 2. Let $L = \lceil \lg(N(1 - 2^{-\Delta})/\Delta) \rceil$. Then

$$\text{EPL}(N, \Delta) \leq NL + \Delta \frac{N - 2^L}{1 - 2^{-\Delta}}.$$

Moreover, for each Δ there exists an infinite subset $\mathcal{N}_\Delta \subseteq \mathcal{N}$ such that for each $N \in \mathcal{N}_\Delta$ there exists a (N, Δ)-tree $T_{(N,\Delta)}$ with

$$\text{EXTERNAL}(T_{(N,\Delta)}) = NL + \Delta \frac{N - 2^L}{1 - 2^{-\Delta}}.$$

Proof. From the analysis in §3.1, it is clear that an upper bound on the external path length of binary trees is given by

$$\text{EPL}(N, \Delta) \leq \max_{x \in \mathcal{N}^+} M(N, \Delta, x).$$

From Lemma 3.2 we get

$$\text{EPL}(N, \Delta) \leq \max_{x \in \mathcal{N}^+} \Phi_{N,\Delta}(x)$$

where

$$\Phi_{N,\Delta}(x) = Nx + \Delta \frac{N - 2^x}{1 - 2^{-\Delta}}.$$

Let $\overline{x} \in \mathcal{N}^+$ be an integer at which $\Phi_{N,\Delta}(x)$ is maximum; i.e,

$$\Phi_{N,\Delta}(\overline{x}) = \max_{x \in \mathcal{N}^+} \Phi_{N,\Delta}(x). \tag{3}$$

As $\Phi_{N,\Delta}(x)$ is a concave function of x, any integer $\overline{x}$ such that

$$\Phi_{N,\Delta}(\overline{x}) \geq \max\left\{\Phi_{N,\Delta}(\overline{x} + 1), \Phi_{N,\Delta}(\overline{x} - 1)\right\}$$

satisfies (3). From the definition of $\Phi_{N,\Delta}(x)$, we obtain that any $\overline{x}$ such that

$$N \frac{1 - 2^{-\Delta}}{\Delta} \leq 2^{\overline{x}} \leq 2N \frac{1 - 2^{-\Delta}}{\Delta}$$

satisfies (3). However, $N(1 - 2^{-\Delta})/\Delta$ is not an integer power of 2; if it were equal to 2^z, for some $z \in \mathcal{N}^+$, then $2^\Delta - 1$ would be a factor of $\Delta 2^{z-\Delta}$, thus of Δ, which is impossible. Hence, the above inequalities uniquely determine $\overline{x}$ as $\overline{x} = \lceil \lg(N(1 - 2^{-\Delta})/\Delta) \rceil$, proving the bound.

Now, we prove that the bound is achievable. Let Δ and x be positive integers such that $2 \leq \Delta \leq x$. Set

$$h_{\Delta,x}(y) = \lg\left(\frac{1-2^{-\Delta}}{\Delta}(2^x + y(2^\Delta - 1))\right).$$

The function $h_{\Delta,x}$ enjoys the following three properties.

(a) $h_{\Delta,x}(1) < x$. Indeed, this inequality is equivalent to

$$2^x > (2^\Delta - 1)^2/(2^\Delta(\Delta - 1) + 1)$$

that is satisfied for $x \geq \Delta$.

(b) $h_{\Delta,x}(2^x - 1) > x$. Indeed, this can be written as

$$2^x > (2^\Delta - 1)^2/(2^\Delta(2^\Delta - 1 - \Delta))$$

that is satisfied for $x \geq \Delta \geq 2$.

(c) $h_{\Delta,x}(y+1) < h_{\Delta,x}(y) + 1$, for $y = 1, 2, \ldots, 2^x - 1$. Indeed, simple algebra shows that it is equivalent to $y > 1 - 2^x/(2^\Delta - 1)$, which holds for $y \geq 1$.

Because of the above three properties, for each $\Delta \geq 2$ and $x \geq \Delta$, there exists some $y_{\Delta,x} \in \{1, 2, \ldots, 2^x - 1\}$ such that $h_{\Delta,x}(y_{\Delta,x}) < x < h_{\Delta,x}(y_{\Delta,x}) + 1$. (Actually, there can be several such $y_{\Delta,x}$'s; for instance, if $\Delta = 5$ and $x = 10$, all integers $s \in [53, 137]$ satisfy $h_{5,10}(s) < 10 < h_{5,10}(s) + 1$.)

For each $\Delta \geq 2$, consider the infinite set $\mathcal{N}_\Delta = \{N : \exists L \geq \Delta \text{ s.t. } N = 2^L + (2^\Delta - 1)y_{\Delta,L}\}$. Fix Δ and $N \in \mathcal{N}_\Delta$, and consider the (N, Δ)-tree T with $2^L - y_{\Delta,L}$ external nodes at level L and $y_{\Delta,L}2^\Delta$ at level $L + \Delta$, where L is such that $N = 2^L + (2^\Delta - 1)y_{\Delta,L}$. First of all, such a tree exists. In fact, as $y_{\Delta,L} \leq 2^L - 1$, it follows $2^L - y_{\Delta,L} > 0$ and the Kraft equality is satisfied. From the definition of $y_{\Delta,L}$ we obtain

$$h_{\Delta,L}(y_{\Delta,L}) \leq L \leq h_{\Delta,L}(y_{\Delta,L}) + 1,$$

and by the definition of the function $h_{\Delta,L}$

$$\lg\left(N\frac{1-2^{-\Delta}}{\Delta}\right) \leq L \leq \lg\left(N\frac{1-2^{-\Delta}}{\Delta}\right) + 1,$$

from which $L = \lceil \lg(N(1-2^{-\Delta})/\Delta) \rceil$ follows.

We now compute $\text{External}(T)$ which turns out to be exactly as in Bound 2. In fact, one has that

$$\begin{aligned}\text{External}(T) &= L(2^L - y_{\Delta,L}) + y_{\Delta,L}2^\Delta(L + \Delta)\\ &= NL + y_{\Delta,L}\Delta 2^\Delta\\ &= NL + \Delta\frac{N - 2^L}{1 - 2^{-\Delta}}.\end{aligned}$$

□

It is clear that Bound 2 is more accurate than Bound 1. In fact, for every $\Delta \geq 2$ and $N \geq \Delta$, (1) is the absolute maximum of the function $\Phi_{N,\Delta}(x) = Nx + \Delta(N - 2^x)/(1 - 2^{-\Delta})$, while (2) is the maximum of the same function subject to the additional constraint $x \in \mathcal{N}^+$.

Example 1. As an example, consider the case $\Delta = 5$. There are binary trees which satisfy Bound 2 with equality for $N = 2^L + 31y$, where $L = 10$ and $y = 53, \ldots, 137$, $L = 11$ and $y = 105, \ldots, 274$, $L = 12$ and $y = 209, \ldots, 549$, $L = 13$ and $y = 418, \ldots, 1099$, and so on for all larger L.

Example 2. By plugging $\Delta = 2$ into Bound 2, we obtain that

$$EPL(N, 2) \leq \left\lfloor (\lceil \lg (3N) \rceil - 3)\, N + \frac{8N - 2^{\lceil \lg(3N) \rceil}}{3} \right\rfloor .$$

Actually, it can be proved that the above bound is tight for all values of N (see [2], [3]).

Example 3. As a simple application of the Bound 2, we derive an upper bound on the internal path length of red-black trees. The class of red-black trees constitutes an important class of balanced binary trees that guarantee a worst-case search cost of $2\log(N + 1)$. The bound we obtain is, up to terms of lower order, equal to the one presented in [1] that was proved to be asymptotically tight.

The bound is obtained by observing that a red-black tree has FRINGE $\leq \log(N + 1)$ and that Bound 2 is increasing in Δ.

BOUND 3. Let T be a red-black tree with N internal nodes. Then

$$\text{INTERNAL}(T) \leq (N + 1)(2\lg(N + 1) - \lg\lg(N + 1) + O(1)).$$

We refer the reader to [3] for a proof of the above bound.

4. Lower bound. In this section we present a lower bound that can be exactly achieved for $N \geq 4$ and $2 \leq \Delta \leq N/2$.

Before dealing with the general case, let us briefly discuss the cases $\Delta = 0$ and $\Delta = 1$ (see [6] for a more complete treatment). The case $\Delta = 0$ is easily settled: a $(N, 0)$-tree exists if and only if N is a power of 2, in which case the N external nodes are located at level $\lg N$ and the tree is unique. On the other hand, a $(N, 1)$-tree exists if and only if N is not a power of 2, in which case it is unique. The external nodes of the *unique* $(N, 1)$-tree are located at levels $\lceil \lg N \rceil - 1$ and $\lceil \lg N \rceil$, and its external path length is $N(\lceil \lg N \rceil + 1) - 2^{\lceil \lg N \rceil}$. The following property of $(N, 1)$-trees can be easily proved using the Kraft equality.

LEMMA 4.1. *If the $(N, 1)$-tree exists and has $c > 0$ external nodes at level $\lceil \lg N \rceil - 1$ then $N + c$ is a power of 2 and, due to $c < N$, none of the integers in $[N, N + c - 1]$ is a power of 2.*

Let us now consider the case $\Delta \leq N/2$ and $N \geq 4$. We start by characterizing the (N, Δ)-trees with minimum external path length. As we will prove in Theorem 4.5, such trees belong to a class of (N, Δ)-trees with a particular structure. Then we compute the external path length of the trees with such a structure and obtain a lower bound by minimizing the so obtained external path lengths.

LEMMA 4.2. *Let T be a (N, Δ)-tree with minimum external path length and configuration $(L; a_0, \ldots, a_\Delta)$. Then*

1. *T has at most two dense levels. Moreover, if T has two dense levels, then these two levels are consecutive.*

2. *If level $L + i$ is dense, for some $1 < i \leq \Delta$, then $a_j = 0$ for all $0 < j < i - 1$.*

Proof. We only prove 1, as the proof of 2 is similar. Assume, by way of contradiction, that T is a tree with two nonconsecutive dense levels and minimum external path length. That is, $a_i, a_k \geq 2$, for $i + 1 < k < \Delta$ (the case in which one of the two dense levels is the last one is similar). Now, consider the tree T' with configuration $(L; a'_0, a'_1, \ldots, a'_\Delta)$ where

$$a'_j = \begin{cases} a_i - 1 & \text{if } j = i, \\ a_j + 3 & \text{if } j = k - 1 = i + 1, \\ a_{i+1} + 2 & \text{if } j = i + 1 \text{ and } j \neq k - 1, \\ a_{k-1} + 1 & \text{if } j = k - 1 \text{ and } j \neq i + 1, \\ a_k - 2 & \text{if } j = k, \\ a_j & \text{otherwise.} \end{cases}$$

T' is still an (N, Δ)-tree and has external path length

$$\text{EXTERNAL}(T') = \text{EXTERNAL}(T) + i + 1 - k < \text{EXTERNAL}(T).$$

Thus T cannot have minimum external path length.

Now, it is immediate that there cannot be more than two intermediate levels with more than one external node even if they are consecutive. In fact, suppose that three levels have more than one external node. Then, obviously, at least two of them must be nonconsecutive. □

Before going any further, let us introduce the following notation. An (N, Δ)-tree of *type* $(L; \geq b_0, a_1, \ldots, a_\Delta)$ has an unspecified, but greater than or equal to b_0, external node at level L; more precisely, it has configuration $(L; a_0, a_1, \ldots, a_\Delta)$ for some $a_0 \geq b_0$. We shall say that an (N, Δ)-tree has type $(L; a_0^{b_0}, a_1^{b_1}, \ldots, a_k^{b_k})$ if it has a_0 external nodes on each of the first b_0 levels, a_1 external nodes on each of the following b_1 levels, and so on; that is, it has configuration

$$(L; \underbrace{a_0, \ldots, a_0}_{b_0 \text{ times}}, \underbrace{a_1, \ldots, a_1}_{b_1 \text{ times}}, \ldots, \underbrace{a_k, \ldots, a_k}_{b_k \text{ times}}).$$

The symbols $\leq a^b$ and $\geq a^b$ denote respectively b levels each of which have no more than a external nodes and no less than a external nodes. As done with the configuration, we will omit L and just say that a tree is of type $(a_0^{b_0}, a_1^{b_1}, \ldots, a_k^{b_k})$ whenever L is either clear from the context or immaterial.

LEMMA 4.3. *Let T be a tree of type $(L; \geq 0^a, \leq 1^b, 2)$. Then if level $L + a'$, with $a' > a$, contains an external node then all levels $L + a' + 1, \ldots, L + a' + b - 1$ contain exactly one external node each; that is, T is of type $(L; \geq 0^{a'-1}, 1^{a+b-a'+1}, 2)$.*

Proof. Consider the grandparent e of the two external nodes at level $L + a + b$. This node has two children, one of which is the internal node which is the parent of the two external nodes at level $L + a + b$. Now, the other child of e cannot be an internal node as this would imply either that level $L + a + b$ has more than two external nodes or that some successive level has some external nodes. Thus, level $L + a + b - 1$ has exactly one external node. We can prove by induction that this is the case also for levels $L + a + b - 2, \ldots, L + a' + 1$, thus proving the lemma. □

We are now ready to characterize completely the configuration of (N, Δ)-trees with minimum external path length. In fact, in Theorem 4.5 we will prove that such a tree must be an (N, Δ, a)-tree defined as follows.

DEFINITION 4.4. For $N \geq 4$, $2 \leq \Delta \leq N - 2$, and $-1 \leq a \leq \min\{\Delta - 2, \lg N - 1\}$ we define an (N, Δ, a)-tree as follows.

1. An $(N, \Delta, -1)$-tree is an (N, Δ)-tree of type $(\geq 1, \geq 0, 1^{\Delta-2}, 2)$.

2. Let $0 \leq a \leq \min\{\Delta - 3, \log N - 2\}$. An (N, Δ, a)-tree is an (N, Δ)-tree of type $(1, 0^a, \geq 0, \geq 0, 1^{\Delta-(a+3)}, 2)$.

3. If $\Delta - 2 \leq \log N - 1$, an $(N, \Delta, \Delta - 2)$-tree is an (N, Δ)-tree of type $(1, 0^{\Delta-2}, \geq 0, \geq 2)$.

Remark. Notice that in the above definition we have not considered the case $a > \log N - 1$. In fact, a tree of type $(L; 1, 0^a, \geq 0, \geq 0, 1^{\Delta-(a+3)}, 2)$ contains at least $2^{L+a+1} - 2^{a+1}$ external nodes. But $L \geq 1$, and thus $2^{a+1} \leq N$, from which $a \leq \lg N - 1$.

Also notice that using the Kraft equality it is easy to see that the (N, Δ, a)-tree, if it exists, is unique. See also the discussion following Lemma 4.6.

THEOREM 4.5. *For $N \geq 4$, $2 \leq \Delta \leq N - 2$, if T is an (N, Δ)-tree with minimum external path length then T is an (N, Δ, a)-tree for some a, $-1 \leq a \leq \min\{\Delta - 2, \lg N - 1\}$.*

Proof. In view of Lemma 4.2, an (N, Δ)-tree with minimum external path length has at most two dense levels and the levels preceding the dense levels (except for the very first one) are all empty.

We consider first the case in which there are two dense levels. If the dense levels are the first two levels, then, because of Lemma 4.3, the tree is of type $(\geq 2, \geq 2, 1^{\Delta-2}, 2)$ and thus is an $(N, \Delta, -1)$-tree. Instead, if the last two levels are dense, then the tree is of type $(1, 0^{\Delta-2}, \geq 2, > 2)$ and is thus an $(N, \Delta, \Delta - 2)$-tree. Finally, if the two dense levels are intermediate then, because of Lemma 4.3, the tree is of type $(1, 0^a, \geq 2, \geq 2, 1^{\Delta-(a+3)}, 2)$, for some a, and thus is an (N, Δ, a)-tree.

Consider now the case in which there exists only one single dense level. Then, if this level is the first, the tree is an $(N, \Delta, -1)$-tree; if it is the last level then the tree is an $(N, \Delta, \Delta - 2)$-tree; otherwise the tree is an (N, Δ, a)-tree for some $0 \leq a \leq \min\{\Delta - 3, \lg N - 2\}$. □

LEMMA 4.6. (a) *For* $0 \leq a \leq \Delta - 3$, *the* $(X, 1)$*-tree* U *with configuration* $(L; a_0, a_1)$, *where* $a_0 \geq 2^{a+1}$, *exists if and only if there exists an* $(X - 2^{a+1} + \Delta - a - 1, \Delta)$*-tree* T *with configuration* $(L - (a + 1); 1, 0^a, a_0 - 2^{a+1}, a_1 - 1, 1^{\Delta-(a+3)}, 2)$.

(b) *The* $(X, 1)$*-tree* U *with configuration* $(L; a_0, a_1)$ *exists if and only if there exists an* $(X + \Delta - 1, \Delta)$*-tree* T *with configuration* $(L; a_0, a_1 - 1, 1^{\Delta-2}, 2)$.

(c) *The* $(X, 1)$*-tree* U *with configuration* $(L; a_0, a_1)$, *where* $a_0 \geq 2^{\Delta-1}$, *exists if and only if there exists an* $(X - 2^{\Delta-1} + 1, \Delta)$*-tree* T *with configuration* $(L - (\Delta - 1); 1, 0^{\Delta-2}, a_0 - 2^{\Delta-1}, a_1)$.

Proof. We prove only part (a) of the lemma, the proofs of parts (b) and (c) being very similar.

For brevity, we denote the configuration of T by $(L - (a + 1); c_0, c_1, \ldots, c_\Delta)$ and $\Delta - a - 2$ by b. First, all the c_i are nonnegative integers if and only if $a_0 \geq 2^{a+1}$. Moreover,

$$\begin{aligned}\sum_{i=0}^{\Delta} \frac{c_i}{2^{L-(a+1)+i}} &= \frac{1}{2^{L-(a+1)}} + \frac{a_0 - 2^{a+1}}{2^L} + \frac{a_1 - 1}{2^{L+1}} + \sum_{i=2}^{b} \frac{1}{2^{L+i}} + \frac{2}{2^{L+b+1}} \\ &= \frac{a_0}{2^L} + \frac{a_1}{2^{L+1}} - \frac{1}{2^{L+1}} + \frac{1}{2^L}\left(\frac{1}{2} - \frac{1}{2^b}\right) + \frac{2}{2^{L+b+1}} \\ &= \frac{a_0}{2^L} + \frac{a_1}{2^{L+1}}.\end{aligned}$$

By using the Kraft equality, the existence of one of the two trees implies that both sums are equal to 1 and thus both trees exist. □

The above lemma implies that, for fixed N, Δ, and a, there exists a unique (N, Δ, a)-tree. In fact, for each a, Lemma 4.6 naturally defines an invertible mapping on trees with $\Delta = 1$ to trees with greater Δ. Therefore, if for some N, Δ, and a there are two (nonisomorphic) (N, Δ, a)-trees, then by inverting the mapping we would construct two (nonisomorphic) trees with $\Delta = 1$ and the same number of external nodes, which is not possible.

The next lemma gives necessary and sufficient conditions for the existence of the (N, Δ, a)-tree and computes its external path length. To this aim, we define the function $F(a, N, \Delta)$ as follows:

$$\begin{aligned}F(a, N, \Delta) = N\Big(\lceil \lg X(a, N, \Delta) \rceil + 1\Big) - 2^{\lceil \lg X(a,N,\Delta) \rceil} + 2^{a+2} + \frac{a(a+1)}{2} \\ + \frac{\Delta(\Delta - 3)}{2} - a\Delta - 2,\end{aligned}$$

where

$$X(a, N, \Delta) = N - \Delta + 2^{a+1} + a + 1.$$

LEMMA 4.7. *For $N \geq 4$, $2 \leq \Delta \leq N - 2$, and $-1 \leq a \leq \min\{\Delta - 2, \log N - 1\}$, the (N, Δ, a)-tree exists if and only if the interval $[X(a, N, \Delta), X(a, N, \Delta) + 2^{a+1} - 1]$ does not contain any power of 2, in which case its external path length is $F(a, N, \Delta)$.*

Proof. We will prove the lemma for $0 \leq a \leq \min\{\Delta - 3, \log N - 1\}$. The cases $a = -1$ and $\Delta - 2 < \log N - 1$ can be proved in a similar way.

Suppose that the interval $[X(a, N, \Delta), X(a, N, \Delta) + 2^{a+1} - 1]$ does not contain any power of 2. Then, by Lemma 4.1 and since $N = X(a, N, \Delta) - 2^{a+1} + \Delta - a - 1$, the unique $(X(a, N, \Delta), 1)$-tree has at least 2^{a+1} external nodes on its first level. Therefore, the hypothesis of part (a) of Lemma 4.6 holds and, thus, the (N, Δ, a)-tree exists. The other implication is proved in a similar way.

To compute the external path length of the (N, Δ, a)-tree, we use Lemma 4.6. The configuration of the (N, Δ, a)-tree, and thus its external path length, is completely determined by the configuration of the corresponding $(X, 1)$-tree. It can be expressed as a function of L, a_0, and a_1, which, in turn, depend only on the value X. We can thus compute the external path length of the (N, Δ, a)-tree, that a tedious computation shows to be exactly $F(a, N, \Delta)$. □

We next study the behavior of the external path length of the (N, Δ, a)-tree, for fixed N and Δ. We will see that the external path length of the (N, Δ, a)-tree is a bitonic function of a; that is, it is nonincreasing up to a certain value $\alpha(\Delta)$ of a and then it is nondecreasing. $\alpha(\Delta)$ is the integer part of the unique solution to the equation $x + 2^{x+1} = \Delta$, $x \in \mathcal{R}$. It is easily seen that $\alpha(\Delta)$ is equal to either $\lfloor \lg \Delta \rfloor - 2$ or $\lfloor \lg \Delta \rfloor - 1$.

LEMMA 4.8. *Let $N \geq 4$, $2 \leq \Delta \leq N - 2$, and $0 \leq a \leq \min\{\Delta - 2, \lg N - 1\}$. If the $(N, \Delta, a - 1)$-tree exists then*

$$F(a, N, \Delta) \begin{cases} \leq F(a - 1, N, \Delta) & \text{if } a \leq \alpha(\Delta), \\ \geq F(a - 1, N, \Delta) & \text{if } a > \alpha(\Delta). \end{cases}$$

Proof. The difference $F(a - 1, N, \Delta) - F(a, N, \Delta)$ is equal to

$$N\Big(\lceil \lg X(a - 1, N, \Delta) \rceil - \lceil \lg X(a, N, \Delta) \rceil\Big) + 2^{\lceil \lg X(a, N, \Delta) \rceil} - 2^{\lceil \lg X(a-1, N, \Delta) \rceil}$$
$$+ \Delta - 2^{a+1} - a.$$

By hypothesis the $(N, \Delta, a - 1)$-tree exists. Therefore, by Lemma 4.7, the interval

$$[X(a - 1, N, \Delta), X(a - 1, N, \Delta) + 2^a - 1]$$

does not contain any power of 2.

If $X(a - 1, N, \Delta) + 2^a$ is a power of 2 then

$$\lceil \lg X(a - 1, N, \Delta) \rceil = \lg(X(a - 1, N, \Delta) + 2^a)$$

and

$$\begin{aligned} \lceil \lg X(a, N, \Delta) \rceil &= \left\lceil \lg\left(X(a - 1, N, \Delta) + 2^a + 1\right) \right\rceil \\ &\qquad \text{(from the definition of } X(a - 1, N, \Delta)) \\ &= \lg(X(a - 1, N, \Delta) + 2^a) + 1. \end{aligned}$$

Therefore,

$$F(a - 1, N, \Delta) - F(a, N, \Delta) = -N + X(a - 1, N, \Delta) + 2^a + \Delta - 2^{a+1} - a = 0,$$

and the lemma is proved.

Let us consider now the case when $X(a-1, N, \Delta) + 2^a$ is not a power of 2. In this case we have

$$\begin{aligned}\lceil \lg X(a, N, \Delta)\rceil &= \left\lceil \lg\left(X(a-1, N, \Delta) + 2^a + 1\right)\right\rceil \\ &= \lceil \lg (X(a-1, N, \Delta))\rceil \\ &\qquad \text{(as the interval } [X(a-1, N, \Delta), X(a-1, N, \Delta) + 2^a] \\ &\qquad \text{does not contain any power of 2.)}\end{aligned}$$

Therefore,

$$F(a-1, N, \Delta) - F(a, N, \Delta) = \Delta - (a + 2^{a+1}).$$

Now, if $a \leq \alpha(\Delta)$ then $a + 2^{a+1} \leq \Delta$ and $F(a, N, \Delta) \leq F(a-1, N, \Delta)$. On the other hand, if $a > \alpha(\Delta)$ then $a + 2^{a+1} > \Delta$ and $F(a, N, \Delta) > F(a-1, N, \Delta)$. □

LEMMA 4.9. *Let $N \geq 4$, $2 \leq \Delta \leq N-2$, and $1 \leq a \leq \min\{\Delta - 2, \lg N - 1\}$. If the $(N, \Delta, a-1)$-tree does not exist and the (N, Δ, a)-tree and the $(N, \Delta, a-2)$-tree exist, then*

$$F(a, N, \Delta)\begin{cases} \leq F(a-2, N, \Delta) & \text{if } a \leq \alpha(\Delta), \\ \geq F(a-2, N, \Delta) & \text{if } a > \alpha(\Delta).\end{cases}$$

Proof. The proof of this lemma is very similar to that of the previous lemma and is omitted. We refer the reader to [3] for a complete proof. □

LEMMA 4.10. *Let $N \geq 4$, $\Delta \leq N/2$, and $\alpha = \alpha(\Delta)$. If the (N, Δ, α)-tree does not exist, then both the $(N, \Delta, \alpha - 1)$-tree and the $(N, \Delta, \alpha + 1)$-tree exist.*

Proof. As the (N, Δ, α)-tree does not exist, then, by Lemma 4.7, the interval

$$[X(\alpha, N, \Delta), X(\alpha, N, \Delta) + 2^{\alpha+1} - 1]$$

contains a power of 2 and the previous power of 2 is in the interval

$$\left[\frac{X(\alpha, N, \Delta)}{2}, \frac{X(\alpha, N, \Delta) + 2^{\alpha+1} - 1}{2}\right].$$

The interval

$$[X(\alpha - 1, N, \Delta), X(\alpha - 1, N, \Delta) + 2^{\alpha} - 1]$$

does not contain any power of 2, and thus the $(N, \Delta, \alpha - 1)$-tree exists, since $X(\alpha - 1, N, \Delta) + 2^{\alpha} - 1 < X(\alpha, N, \Delta)$ and

$$X(\alpha - 1, N, \Delta) > \frac{X(\alpha, N, \Delta) + 2^{\alpha+1} - 1}{2}.$$

The first inequality follows from the definition of $X(a, N, \Delta)$, while the second is equivalent to $N - \Delta > 2^{\alpha+1} - \alpha$. As $\Delta \leq N/2$ and $\alpha \leq \log \Delta - 1$, the above inequality holds, thus proving that the $(N, \Delta, \alpha - 1)$-tree exists.

The existence of the $(N, \Delta, \alpha + 1)$-tree is established in a similar way. □

We can summarize the above lemmata by saying that if the (N, Δ, α)-tree exists, that is, if the interval $[X(\alpha, N, \Delta), X(\alpha, N, \Delta) + 2^{\alpha+1} - 1]$ does not contain any power of 2, then *epl*$(N, \Delta) = F(\alpha, \Delta, N)$. When the (N, Δ, α)-tree does not exist, then, by Lemma 4.10, both the $(N, \Delta, \alpha - 1)$-tree and the $(N, \Delta, \alpha + 1)$-tree exist and the minimum path length is obtained by either one.

We have thus proved the following:

BOUND 4. For all $N \geq 4$ and $\Delta \leq N/2$,

$$epl(N, \Delta) = \begin{cases} F(\alpha, \Delta, N) & \text{if } [X(\alpha, N, \Delta), X(\alpha, N, \Delta) + 2^{\alpha+1} - 1] \\ & \quad \text{does not contain any power of 2,} \\ F(\alpha + 1, \Delta, N) & \text{if } F(\alpha + 1, \Delta, N) \leq F(\alpha - 1, \Delta, N), \\ F(\alpha - 1, \Delta, N) & \text{otherwise,} \end{cases}$$

where α is the integer part of the (unique) solution of the equation $x + 2^{x+1} = \Delta$,

$$X(a, N, \Delta) = N - \Delta + 2^{a+1} + a + 1,$$

and

$$F(a, N, \Delta) = N\Big(\lceil \lg X(a, N, \Delta) \rceil + 1\Big) - 2^{\lceil \lg X(a,N,\Delta) \rceil} + 2^{a+2}$$
$$+ \frac{a(a+1)}{2} + \frac{\Delta(\Delta - 3)}{2} - a\Delta - 2.$$

Acknowledgments. The second author would like to thank Professor Harry Lewis for his suggestions. We also thank Marios Mavronicolas for a careful reading of the manuscript.

REFERENCES

[1] H. CAMERON AND D. WOOD, *The Internal Path Length of Red-Black Trees*, Research Report CS-89-50, Computer Science Department, University of Waterloo.

[2] A. DE SANTIS AND G. PERSIANO, *Tight bounds on the path length of binary trees*, in "STACS 91", LNCS, C. Choffrut and M. Jantzen, eds., Vol. 480, Springer-Verlag, New York, pp. 478–487.

[3] ———, *Tight Upper and Lower Bounds on the Path Length of Binary Trees*, Tech. report TR-07-91, Aiken Computation Laboratory, Harvard University, February 1991.

[4] R. W. HAMMING, *Coding and Information Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1980.

[5] R. KLEIN AND D. WOOD, *On the path length of binary trees*, J. Assoc. Comput. Mach., 36 (1989), pp. 280–289.

[6] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3 *Sorting and Searching*, Addison–Wesley, Reading, MA, 1973.

[7] J. NIEVERGELT AND C. K. WONG, *Upper bounds for the total path length of binary trees*, J. Assoc. Comput. Mach., 20 (1973), pp. 1–6.

[8] W. SPECHT, *Zur Theorie der Elementaren Mittel*, Math. Z., 74 (1960), pp. 91–98.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 24–44, February 1994

UNIQUE BINARY-SEARCH-TREE REPRESENTATIONS AND EQUALITY TESTING OF SETS AND SEQUENCES*

RAJAMANI SUNDAR[†] AND ROBERT E. TARJAN[‡]

Abstract. This paper studies the problem of representing sets over an ordered universe by unique binary search trees, so that dictionary operations can be performed efficiently on any set. Although efficient randomized solutions to the problem are known, its deterministic complexity has been open. The paper exhibits representations that permit the execution of dictionary operations in optimal deterministic time when the dictionary is sufficiently sparse or sufficiently dense. The results demonstrate an exponential separation between the deterministic and randomized complexities of the problem.

Unique representations are applied to obtain efficient data structures for maintaining a dynamic collection of sets/sequences under queries that test the equality of a pair of objects. The data structure for set equality testing tests equality of sets in constant time and processes set updates in $O(\log m)$ amortized time and $O(\log m)$ space, where m denotes the total number of updates performed. It is based on an efficient implementation of cascades of CONS operations on uniquely stored S-expressions. The data structure for sequence equality testing tests equality of sequences in constant time and processes updates in $O(\sqrt{n}\log m + \log m)$ amortized time and $O(\sqrt{n})$ amortized space where n denotes the length of the sequence that is updated and m denotes the total number of updates performed.

Key words. data structures, programming languages, unique representation, dictionary, binary search tree, equality testing, sets, sequences, S-expressions

AMS subject classifications. 68P05, 68Q20, 68Q25, 68R05

1. Introduction.

A *unique representation* of an abstract data type (e.g., a dictionary) by a concrete data type (e.g., a binary search tree) consists of a one-to-one mapping of values of the abstract data type (ADT) into values of the concrete data type (CDT) and an implementation of the ADT operations by the CDT that *preserves* this mapping. A CDT implementation of an ADT operation that changes an ADT value v to a value w *preserves* this mapping if the implementation changes the image of v into the image of w. In this paper we are interested in devising unique representations of dictionaries by binary search trees that support dictionary operations (i.e., searches, insertions, and deletions of elements) efficiently and in applying our unique representations to equality testing of sets and sequences.

Unique representations arise in the contexts of incremental evaluation and implementation of high-level programing techniques. Incremental evaluation [14], [15] is the technique of efficiently updating the value of a function when the input changes. A simple idea to speed up incremental evaluation is to remember the results of previous function calls and thereby avoid recomputation. If the input domain is uniquely represented and input data objects are *stored uniquely*, then it becomes easy to check whether the function has already been evaluated on a given input. In this context *unique storage* of data objects means that instances of the same data object created at different times are all represented physically by the same storage structure. Modern programming languages, such as SETL, support high-level data types, such as sets and sequences, and permit testing whether two such objects are equal. Equality testing

*Received by the editors October 22, 1990; accepted for publication (in revised form) June 24, 1992. A preliminary version of this paper appeared in the *Proceedings of the Twenty-Second Annual Symposium on Theory of Computing*, 1990, pp. 18–25.

[†] Department of Computer Science, Princeton University, Princeton, New Jersey 08544-2087. Research by this author at Bellcore, DIMACS (National Science Foundation Science and Technology Center for Discrete Mathematics and Theoretical Computer Science), and the Courant Institute was partially supported by an IBM graduate fellowship and National Science Foundation grants CCR-8702271, CCR-8902221, CCR-8906949, and STC88-09648 and by Office of Naval Research grant N00014-85-K-0046.

[‡] Department of Computer Science, Princeton University, Princeton, New Jersey 08544-2087. The work of this author was partially supported by National Science Foundation grants DCR-8605962 and STC88-09648 and by Office of Naval Research grant N00014-91-J-1463.

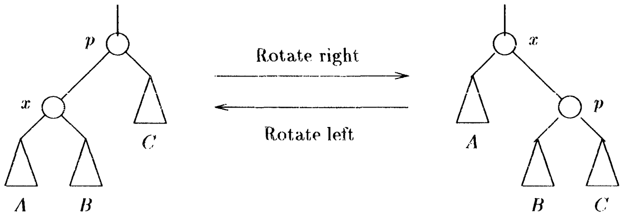


FIG. 1. *Rotation in a binary tree.*

can be implemented in constant time by devising a unique representation for the data type and uniquely storing concrete data values.

We define some basic concepts. A *dictionary* is a set of items selected from a totally ordered universe on which membership queries and update operations that insert or delete items can be performed. We can represent a dictionary by a binary tree containing one item per node, with the items arranged in symmetric order: each item in the tree is larger than the items in its left subtree and smaller than the items in its right subtree. This data structure is called a *binary search tree.* A binary search tree is represented in storage as a collection of records, one per node, with a pointer to the record corresponding to the root. Each record has a key field, left and right child pointers, and a fixed number of additional information and pointer fields. A *rotation* of an edge $[x, p]$ in a binary tree, where p is the parent of x, is a transformation that makes x the parent of p by transferring one of the subtrees of x to p; see Fig. 1.

We review previous work on unique representations. Snyder [18] studied unique representations of sets over an ordered universe by search trees so that, for each set size, all sets of that size have the same underlying tree. Search trees can be updated in various ways; in Snyder's work search trees are updated by changing the pointers of a set of nodes. For his class of unique representations Snyder showed that $\Theta(\sqrt{n})$ time is necessary and sufficient to perform a dictionary operation, where n is the dictionary size. Munro and Suwanda [13] studied how to implicitly represent sets over an ordered universe by an array so that, for each set size, all sets of that size satisfy a unique partial order on the array locations. They showed that $\Theta(\sqrt{n})$ time is necessary and sufficient to perform dictionary operations by using their representations. Recent work has focused on randomized unique representations. Pugh [14] and Pugh and Tietelbaum [15] gave efficient randomized unique representations for sets, sequences, and other abstract data types. Aragon and Seidel [3] gave a randomized binary-search-tree data structure that has the unique representability property and supports dictionary operations in $O(\log n)$ randomized time.

This paper studies the deterministic complexity of uniquely representing a dictionary over an ordered universe U by a binary search tree. We explore the following methods of updating a binary search tree during an insertion or deletion:

1. Performing a sequence of rotations: In this method we update the binary search tree representing the dictionary by using the traditional algorithms for binary-search-tree insertion and deletion [11], but we permit arbitrary rotations to be performed on the tree before and after so that the final tree correctly represents the updated dictionary.

2. Nondestructive updating through CONS operations: In this method we perform a sequence of dictionary operations on the null tree, and we maintain all the trees created by the operations performed so far and their subtrees. An update operation constructs a new tree by accessing some of its subtrees from the existing collection of trees and creating the remaining subtrees from the accessed subtrees, proceeding bottom-up. Also, an update operation may create any number of additional trees that may be needed in the future. A

CONS operation is used to create a new tree, given its left and right subtrees and root item. This operation parallels the CONS operation on S-expressions occurring in LISP [1]. A unique binary-search-tree representation that can be updated efficiently by means of CONS operations can be used to devise a data structure for maintaining a collection of sets/sequences under equality tests. In order to perform equality tests quickly, we assign *signatures* to binary search trees representing sets/sequences so that identical trees get identical signatures. Since we are using a unique representation, two sets/sequences are equal if and only if their binary search trees are identical, in other words, if their signatures are identical. A CONS operation computes the signature of a new tree it creates by matching it in the current collection of trees. If there is an identical tree in the collection, its signature is assigned to the new tree; otherwise, a new signature is assigned. CONS operations are the expensive part of the data structure since they have to search in the collection of trees. At present we do not know whether CONS operations can be implemented in constant time; later in this section we discuss the fastest known methods of implementing CONS operations. In our unique representation we define the cost of an update operation as the number of CONS operations performed instead of as the actual time taken. This is reasonable since CONS operations are the bottleneck in equality testing data structures and since we do not know their exact complexity.

For each updating method we seek representations that permit efficient implementation of the dictionary by using that method. Since $2n - 2$ rotations suffice to transform any n-node binary tree into any other n-node binary tree (see [7], [17]), any unique binary-search-tree representation supports dictionary operations in $O(n)$ time when rotations are used. We provide a representation that supports searches in $O(\log n)$ time and updates in $O(\sqrt{n})$ CONS operations.

We prove lower bounds on the complexity of any unique binary-search-tree representation that apply to sparse dictionaries (sets that are small compared to the universe). The lower bounds are derived for the following cost metric. The *cost* of an update using one of the aforementioned methods is, respectively, the number of rotations or the number of CONS operations. The *cost* of searching for an item in a binary search tree is 1 plus the depth of the item in the tree. Under the assumption that the dictionary is sparse, we show that the $O(n)$ and $O(\sqrt{n})$ upper bounds mentioned in the last paragraph are optimal. Our proof technique also extends Snyder's $\Omega(\sqrt{n})$ lower bound in the pointer-changes model to unique representations in which sets of equal cardinality need not have the same underlying tree. In light of Aragon and Seidel's representation, these lower bounds imply an exponential separation between the deterministic and randomized complexities of the problem. The sparseness assumption in the lower bounds is essential since we construct a representation that requires $O(\log |U|)$ cost per dictionary operation using any update strategy. This representation is optimal on dense dictionaries.

We apply unique representations to the problems of equality testing of sets and sequences.

Set equality testing problem: Maintain a collection of sets over an ordered universe under the operations i) EQUAL(S, T)—test whether sets S and T are equal, ii) INSERT(S, x, T)—create a new set T by inserting element x into set S, and iii) DELETE(S, x, T)—create a new set T by deleting element x from set S. Initially, the collection consists of only the empty set.

Sequence equality testing problem: Maintain a collection of sequences over an ordered universe under the operations i) EQUAL(S, T)—test whether sequences S and T are equal, ii) INSERT(S, i, x, T)—create a new sequence T by inserting element x into sequence S as the ith element between old positions $i - 1$ and i, and iii) DELETE(S, i, T)—create a new set T by deleting the ith element of sequence S. Initially, the collection consists of only the null sequence.

For both problems we are interested only in data structures that test equality in constant time. Wegman and Carter [20] proposed a randomized signature-based scheme for set equality testing that requires only constant time per operation but errs with a small probability. Pugh [14] and Pugh and Tietelbaum [15] gave randomized solutions to both problems that require $O(\log n)$ expected time and $O(\log n)$ expected space per update operation, where n denotes the size of the set or sequence. Their data structures also support more powerful operations (for instance, union and intersection of sets) efficiently. For sequence equality testing, however, the logarithmic bound is valid only if the sequences do not contain repeated elements. Yellin [23] provided a deterministic solution for set equality testing that requires $O(\log^2 m)$ time and $O(\log m)$ space per update operation, where m denotes the total number of updates.

We devise a data structure for set equality testing with $O(\log m)$ amortized time and $O(\log m)$ space per update. The data structure is based on a solution to a more fundamental problem involving S-expressions. S-expressions [1] constitute the staple data type of the programming language LISP. An *S-expression* is either an *atom* (signifying a number or a character string) or a pair of S-expressions. An atom is represented by a node in storage, and a pair is represented by a node with left and right pointers that point to nodes representing the component S-expressions. We store S-expressions uniquely, i.e., all instances of an S-expression are represented by a single node. CONS(s_1, s_2) returns the S-expression $(s_1.s_2)$. A *casade* of CONS operations is a sequence of CONS operations such that the result of each CONS operation is an input to the next CONS operation. For instance,

$$\begin{aligned} s_1 &:= \textsc{Cons}(s_0, t_0), \\ s_2 &:= \textsc{Cons}(t_1, s_1), \\ s_3 &:= \textsc{Cons}(t_2, s_2), \\ &\ \vdots \\ s_f &:= \textsc{Cons}(s_{f-1}, t_{f-1}) \end{aligned}$$

is a cascade of f CONS operations. The S-expression problem in question is to devise a data structure for efficiently implementing cascades of CONS operations on uniquely stored S-expressions.

Unique storage of S-expressions makes CONS operations expensive. Single CONS operations can be implemented in $O(\sqrt{\log F})$ time and $O(1)$ amortized space or, alternatively, in $O(1)$ time and $O(F^\epsilon)$ space, where F denotes the total number of CONS operations performed and ϵ is any positive constant. This implementation is based on Willard's data structure [22] for maintaining a dictionary in a small universe. Universal hashing [4] and dynamic perfect hashing [8] offer alternative implementations that require $O(1)$ randomized amortized time and $O(1)$ amortized space per CONS operation.

We develop a data structure that performs a cascade of f CONS operations in $O(f + \log m_c)$ amortized time, where m_c denotes the total number of cascades performed. The total space used is proportional to the number of distinct S-expressions present. When sets are represented by binary tries, an update operation translates into a cascade of at most $\log m$ CONS operations and requires $O(\log m)$ amortized time when this data structure is used. Many list-oriented functions in functional languages (LISP, for instance) involve cascades of CONS operations and can be implemented efficiently by using this method. Function APPEND is a typical example:

$$\begin{aligned} &\textsc{Append}([v_1, v_2, \ldots, v_k], [w_1, w_2, \ldots, w_l]) \equiv \\ &s_1 \quad := \textsc{Cons}(v_k, [w_1, w_2, \ldots, w_l]), \\ &s_2 \quad := \textsc{Cons}(v_{k-1}, s_1), \\ &\quad\ \vdots \\ &\textbf{Result} := \textsc{Cons}(v_1, s_{k-1}). \end{aligned}$$

Sequence equality testing can be solved in $O(n+\log m)$ amortized time and $O(1)$ space per update operation by maintaining sequences in a lexicographic search tree [16]. Here n denotes the length of the sequence that is updated and m denotes the total number of updates. Our solution to this problem requires $O(\sqrt{n \log m} + \log m)$ amortized time and $O(\sqrt{n})$ amortized space per update operation. When sequences are free from repetitions, sequence equality testing is equivalent to set equality testing. This follows from the representation of sets by sorted sequences and the representation of sequences by sets of ordered pairs of adjacent elements. Therefore, sequence equality testing becomes truly hard only when sequences have repetitions.

Recently, Andersson and Ottmann [2] have studied unique representation of a dictionary by directed graphs with nodes of bounded outdegree. Their class of unique digraph representations encompasses Snyder's and our classes of unique representations. They prove a tight bound of $\Theta(n^{1/3})$ on the time required by dictionary operations using their unique digraph representations. Unfortunately, their unique representation cannot be used to test equality of sets/sequences in constant time because the representation is updated through arbitrary pointer changes instead of through CONS operations; in order to transform a unique representation into an equality-testing data structure it should be updated by using CONS operations alone.

It might be possible to solve the sequence equality problem efficiently by devising a unique representation of sequences by bounded-outdegree directed acyclic graphs (DAG's) that can be quickly updated through CONS operations when elements are inserted into or deleted from sequences. It is natural to ask whether there is a unique DAG representation of sequences that can be updated in $O(\text{polylog}(n))$ CONS operations; n denotes the length of the updated sequence. We study the complexity of unique DAG representation of sequences in which all sequences of the same length are represented by the same underlying DAG and show that update operations in these representations require $\Omega(\sqrt{n}/\log n)$ CONS operations. Hence this restricted class of unique DAG representations cannot lead to a fast data structure for sequence equality testing. Our lower bound extends to arbitrary unique DAG representations of sequences if the underlying universe is sufficiently large.

The paper is organized as follows. Section 2 describes the results on unique binary-search-tree representations. Sections 3 and 4, respectively, describe the data structures for set and sequence equality testing. Section 5 describes the lower-bound result for unique DAG representations of sequences. Section 6 poses open problems.

2. Unique binary-search-tree representations.

In this section we describe the optimal representations for sparse and dense dictionaries and establish the lower bounds for representing sparse dictionaries.

2.1. An optimal representation for sparse dictionaries.

We describe a representation that allows search in $O(\log n)$ time and permits updating the binary search tree in $O(\sqrt{n})$ CONS operations. We represent the dictionary by an almost complete binary search tree. Specifically, if $n = 2^{i_1} + 2^{i_2} + \cdots + 2^{i_k}$, such that $i_1 > i_2 > \cdots > i_k$, then the binary search tree representing an n-element dictionary comprises a k-node right path, with a sequence of k complete binary trees of respective sizes $2^{i_1} - 1, 2^{i_2} - 1, \ldots, 2^{i_k} - 1$ hanging off of it; see Fig. 2a.

In order to be able to update the tree efficiently, we store some additional information. Define a *j-run* to be a subset of $2^j - 1$ adjacent elements in the dictionary; a *j-tree* is a complete binary search tree representing a j-run. For each $j = 1, 2, \ldots, \lfloor \log n/2 \rfloor$ we store the j-trees of the dictionary in a sorted list. This list is simply a doubly linked list of the roots of the j-trees; the subtrees of the j-trees are represented by pointers to the roots of these subtrees in the list of $(j-1)$-trees; see Figs. 2b and 2c. When n has the form $2^{2i} - 2^i + k$, where $1 \le k < 2^i$, in addition to the preceding lists we also store a sorted list of i-trees

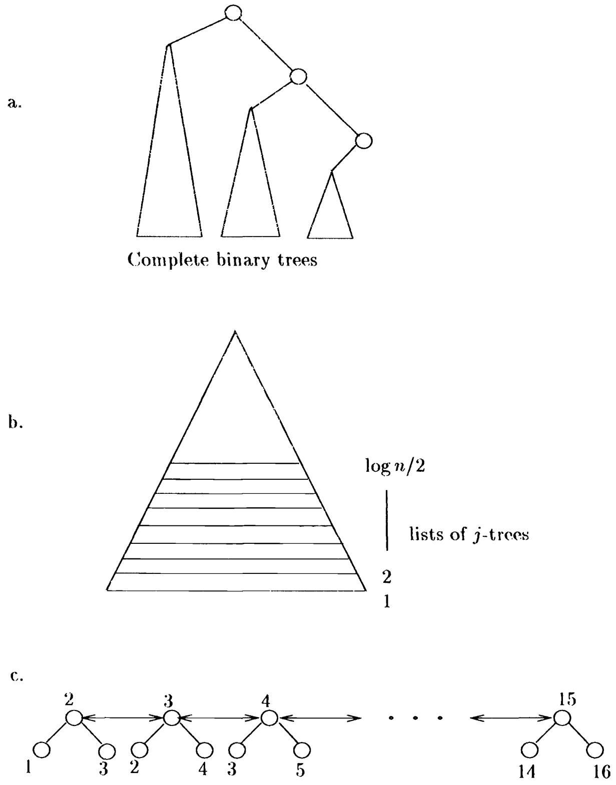


FIG. 2. *a. An almost complete binary tree having $2^{i_1} + 2^{i_2} + 2^{i_3}$ elements, where $i_1 > i_2 > i_3$. b. Lists of j-trees. c. List of 2-trees in the representation of set* $\{1, 2, \ldots, 16\}$.

($i = \lfloor \log n/2 \rfloor + 1$) corresponding to the first $k2^i$ i-runs. When an insertion causes n to increase from $2^{2i} - 1$ to 2^{2i} (causing $\lfloor \log n/2 \rfloor$ to increase by 1), this list becomes the list of $\lfloor \log n/2 \rfloor$-trees. This completes our representation; the storage used is $O(n \log n)$.

It is obvious that this representation supports search in $O(\log n)$ time. To insert a new element x we first locate the position of x in the list of 1-trees by searching the tree from top to bottom, then update the lists of trees, proceeding bottom-up, and then create the rest of the tree from trees in the updated lists. Updating the list of j-trees, for any j, involves locating and deleting at most $2^j - 2$ j-trees corresponding to old j-runs that contain the predecessor as well as the successor of x and inserting in their place at most $2^j - 1$ j-trees corresponding to new j-runs that contain x. We locate the j-trees to be replaced by starting at the j-tree encountered while searching for element x in the tree and traversing the list of j-trees left and right; this takes $O(2^j)$ time. We create a new j-tree by performing a CONS operation on two $(j-1)$-trees that are its subtrees and its root item. If n is of the form $2^{2i} - 2^i + k$, where $1 \leq k < 2^i$, we also extend the size of the list of i-trees from $k2^i$ to $(k+1)2^i$ by creating at most 2^i new i-trees. The list of j-trees can be updated in $O(2^j)$ time plus $O(2^j)$ CONS operations once the list of $(j-1)$-trees has been updated. The total cost of updating the lists of trees is at most $O(2 + 2^2 + \cdots + 2^{\lfloor \log n/2 \rfloor + 1}) = O(\sqrt{n})$. To create the rest of the tree we need to create the nodes of the tree at levels $\lfloor \log n/2 \rfloor + 1, \lfloor \log n/2 \rfloor + 2, \ldots, \lfloor \log n \rfloor + 1$ and the nodes on the right path of the tree. We traverse the nodes of the old tree at these levels, not on the right path, in postorder and replace them with nodes of the new tree by using

CONS operations. This gives us the subtrees of the right path of the new tree at the upper-half levels. We locate the subtrees of the right path at the lower-half levels in the updated lists of trees by traversing these lists top-down and left-to-right in staircase fashion. The subtrees of the right path of the new tree are all available now, so we create this path from bottom to top. It is easy to see that only $O(\sqrt{n})$ CONS operations are performed during an insertion and only $O(\sqrt{n})$ extra work is performed in addition to the CONS operations; the storage used is $O(\sqrt{n})$. Deletion in analogous.

2.2. Lower bounds for sparse dictionaries. Our lower bounds are proved by using Ramsey's theorem [10], [12].

RAMSEY'S THEOREM. *Let n, k, and $s \geq n$ be arbitrary positive integers, and let U be an arbitrary set. There exists a number $R_n(k, s)$ with the following property: if $|U| \geq R_n(k, s)$, then for any partition of the n-subsets of U into k classes there is an s-subset S all of whose n-subsets lie in a single class.*

It is known that $R_n(k, s) = T_n(O(\log(ks)))$ [10], [12], where $T_n(x)$ is the tower function

$$\left.2^{2^{\cdot^{\cdot^{2^x}}}}\right\}n.$$

First, we state a lower bound for updating the tree by means of rotations. Let $b_n = \binom{2n}{n}/(n+1)$ denote the number of distinct n-node binary trees [11]. Let n and d be positive integers, and let U be an ordered universe of size at least $R_n(b_n, n+1)$. For any unique binary-search-tree representation of subsets of U, we have the following trade-off between search and update times.

THEOREM 1. *If the n-subsets of U are represented by binary search trees of height at most d, then there is an n-subset on which an update operation requires $\Omega(n - 2d)$ rotations.*

The theorem implies that a dictionary operation requires $\Omega(n)$ time in the worst case if the dictionary is sufficiently sparse and rotations are used to update the tree. It is easy to show that the trade-off given by the theorem is optimal, up to a constant factor. The Ramsey number $R_n(b_n, n+1)$ is at most $T_{n+1}(cn)$, for some constant c. Therefore, the trade-off is valid when the dictionary size is at most $(1 - o(1)) \log^* |U|$.

The next theorem gives the lower bound on the cost of updating the tree by using CONS operations.

THEOREM 2. *Let n and $m \geq 2n$ be positive integers, and let U be an ordered universe of size at least $R_n(b_n, m)$. For any unique binary-search-tree representation of subsets of U there is a sequence of m update operations that involves only subsets of size at most n and causes $\Omega(m\sqrt{n})$ CONS operations to be performed.*

Snyder [18] showed a lower bound of $\Omega(\sqrt{n})$ on the worst-case cost of a dictionary operation when updates are implemented through pointer changes. His lower bound is restricted to the class of unique binary-search-tree representations in which all sets of equal cardinality have the same underlying tree. When the dictionary is sufficiently sparse, the lower bound can be extended to the class of all unique binary-search-tree representations by using Ramsey's theorem. The way to accomplish this is illustrated by the proofs of Theorems 1 and 2.

Proof of Theorem 1. By Ramsey's theorem there is a subset $S = \{x_1, x_2, \ldots, x_{n+1}\}$ of U all of whose n-subsets are represented by a single underlying binary tree, say, B. We claim that at least $n - 1 - 2d$ rotations are necessary to transform set $S_1 = \{x_1, x_2, \ldots, x_n\}$ into set $S_2 = \{x_2, x_3, \ldots, x_{n+1}\}$. The claim is proved by using Wilber's method [21] of deriving lower bounds on rotations in binary trees.

Consider an n-node binary tree T whose nodes are labeled from 1 to n in symmetric order. An interval $[i, j]$ of nodes of T is a *block* of T. Any block $[i, j]$ of T induces a binary tree,

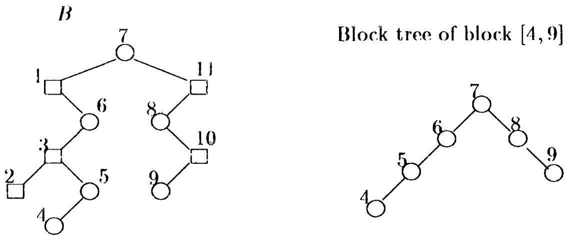


FIG. 3. *Block tree.*

called the *block tree* of block $[i,\ j]$, obtained by contracting all edges of T having an endvertex outside the block; see Fig. 3.

A rotation on T propagates into a rotation on a block tree of T only if both nodes of the rotation lie in the block tree; otherwise, the block tree is unaffected. Let $T_{\min}$ denote the binary tree obtained by deleting the minimum element of T (equivalently, the block tree of block $[2, n]$ in T). Define $T_{\max}$ similarly. Let $d_{\min}(T)$ and $d_{\max}(T)$ denote, respectively, the depths of the minimum and maximum elements of T.

LEMMA 1. *For any binary tree T, at least $|T| - 1 - d_{\min}(T) - d_{\max}(T)$ rotations are necessary to transform $T_{\min}$ into $T_{\max}$.*

Proof. The lemma is proved by induction on $|T|$.

Basis. $|T| \leq 2$: Easy.

Induction step. $|T| \geq 3$: Let x denote (the symmetric order number of) the node in T that is closest to the root and that is not the maximum or the minimum element. Let L and R denote, respectively, the block trees of blocks $[1, x]$ and $[x, |T|]$ in T. Divide the nodes of $T_{\min}$ and $T_{\max}$ into a left block $[1, |L| - 1]$ and a right block $[|L|, |T| - 1]$. Then the left block trees of $T_{\min}$ and $T_{\max}$ are respectively, $L_{\min}$ and $L_{\max}$; see Fig. 4.

By the inductive hypothesis at least $|L| - 1 - d_{\min}(L) - d_{\max}(L)$ left block rotations must be performed to transform $L_{\min}$ into $L_{\max}$. Similarly, at least $|R| - 1 - d_{\min}(R) - d_{\max}(R)$ right block rotations are needed to transform the right block tree $R_{\min}$ of $T_{\min}$ into the right block tree $R_{\max}$ of $T_{\max}$. Hence the total number of rotations necessary to transform $T_{\min}$ into $T_{\max}$ is at least $|T| - 1 - d_{\min}(L) - d_{\max}(L) - d_{\min}(R) - d_{\max}(R)$. The rest of the proof breaks up into three cases according to the position of x in T.

Case 1. x is the root of T: In this case we have

$$d_{\min}(L) = d_{\min}(T), \quad d_{\max}(L) = d_{\min}(R) = 0, \quad d_{\max}(R) = d_{\max}(T).$$

The inductive hypothesis follows from these relations.

Case 2. x is the only child of the root: The root must be either the minimum or the maximum element. Without loss of generality, assume that the root is the minimum element. We have

$$d_{\min}(L) = d_{\min}(R) = d_{\min}(T) = 0, \quad d_{\max}(L) = 1, \quad d_{\max}(R) = d_{\max}(T) - 1.$$

The inductive hypothesis follows.

Case 3. x is the only grandchild of the root: The minimum and the maximum elements of T constitute the root and its only child. Without loss of generality, let the minimum element be the root and let the maximum element be its child. We have

$$d_{\min}(L) = d_{\min}(R) = d_{\min}(T) = 0, \quad d_{\max}(L) = d_{\min}(R) = d_{\max}(T) = 1.$$

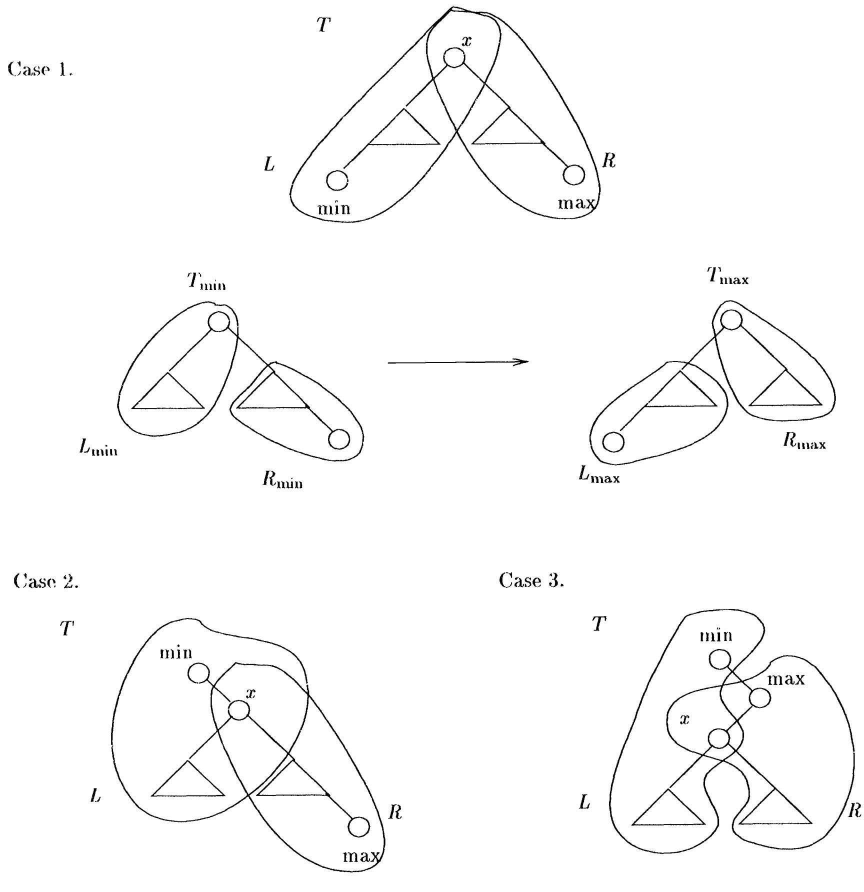


FIG. 4.

Since the roots of T_{min} and T_{max} are from different blocks, in addition to left and right block rotations, at least one rotation involving nodes from both blocks must be performed in transforming T_{min} into T_{max}. The inductive hypothesis follows from this observation and the preceding relations. □

When set S_1 is transformed into set S_2, the block tree induced by subset $\{x_2, x_3, \ldots, x_n\}$ is transformed from B_{min} into B_{max}. By Lemma 1 it follows that at least $n - 1 - 2d$ rotations are required to transform S_1 into S_2.

Consider transforming S_1 into S_2 by deleting x_1 and inserting x_{n+1}. The insertion of x_{n+1} into set $\{x_2, x_3, \ldots, x_n\}$ and the deletion of x_{n+1} from set S_2 are inverse transformations and require the same number of rotations. Hence $\Omega(n - 2d)$ rotations are necessary for either the deletion of x_1 from set S_1 or the deletion of x_{n+1} from set S_2. □

Proof of Theorem 2. By Ramsey's theorem there is a subset $S = \{x_1, x_2, \ldots, x_m\}$ of U all of whose n-subsets are represented by the same underlying binary tree. Let T denote an n-subset of S, to be specified later. The lower-bound construction inserts the elements of T into the null tree, one by one, and then performs $m - n$ updates on the resulting tree by repeating the following cycle of operations sufficiently many times:

1. Repeat $\sqrt{n} - 1$ times: Delete the minimum element of the tree and insert a fresh element from S that is larger than the maximum element of tree. Throughout this construction, by fresh element we mean an element that has never been in the tree before.

2. Replace the elements of the tree with symmetric order numbers $\sqrt{n}, 2\sqrt{n}, \ldots, n$ by fresh elements from S that occupy the same symmetric order positions in the tree.

This completes the construction. We now define set T and the fresh elements inserted into the tree during the construction. Execute the construction, treating all elements used as indeterminants, and determine their partial ordering. Next, assign specific values to these elements from set S to fit the partial order.

An iteration of the cycle performs $4\sqrt{n} - 2$ update operations on the tree (the first step performs $2(\sqrt{n} - 1)$ update operations; the second step performs $2\sqrt{n}$ update operations). Since $m - n \geq m/2$ update operations are performed during all iterations of the cycle, it follows that the number of iterations is at least $m/(8\sqrt{n})$. We claim that an iteration of the cycle creates n new trees. This would imply that at least $m\sqrt{n}/8$ new trees are created during the entire construction and would prove the theorem. Let us prove the claim. For $i = 1, 2, \ldots, \sqrt{n}$ let v_i denote the highest node of the tree with symmetric order number in the interval $[(i-1)\sqrt{n} + 1, i\sqrt{n}]$. Consider an iteration of Step 1 of the cycle. The subtrees of nodes $v_1, v_2, \ldots, v_{\sqrt{n}}$ after the iteration are all different from the subtrees of the tree at any time before the iteration; this is caused by the leftward movement of the fresh elements inserted during the preceding cycle. Since Step 1 is iterated $\sqrt{n} - 1$ times, it follows that $n - \sqrt{n}$ new trees are created during this step. Step 2 creates $\sqrt{n}$ new trees, so it follows that n new trees are created during an iteration of the cycle. The proof of the claim is complete; Theorem 2 follows. □

2.3. An optimal representation for dense dictionaries. Aragon and Seidel's representation [3] creates trees of height at least $\sqrt{|U|}$ in the worst case. They choose a static, random priority for each element in the universe and represent a set of items by the binary search tree that is heap-ordered according to priorities. It is well known [12] that any sequence of $p^2 + 1$ numbers contains a monotone subsequence of length at least $p + 1$. If we consider the longest monotone subsequence of the sequence of priorities of elements in the universe, then the representation of the set corresponding to this subsequence has height at least $\sqrt{|U|}$. Therefore, Aragon and Seidel's representation requires $\Omega(\sqrt{|U|})$ time in the worst case to perform a dictionary operation. Furthermore, since it is possible to assign priorities to elements of the universe so that the longest monotone subsequence has length at most $\sqrt{|U|} + 1$, there is a static priority representation in which trees have height $O(\sqrt{|U|})$. This representation allows dictionary operations to be performed in $O(\sqrt{|U|})$ time.

We describe a more efficient representation, based on binary tries [11], that allows dictionary operations to be performed in $(O \log |U|)$ time. Assume that $U = [1, 2^p]$, and let S denote the set being represented. If $S \subseteq [1, 2^{p-1}]$, then S is represented in U by the tree representing it in the universe $[1, 2^{p-1}]$. Otherwise, let $r = \min(S \bigcap [2^{p-1} + 1, 2^p])$. Then, r is the root of the tree representing S, and its left and right subtrees are, respectively, the trees representing subsets $S \bigcap [1, 2^{p-1}]$ and $S \bigcap [r + 1, 2^p]$ in the universes $[1, 2^{p-1}]$ and $[2^{p-1} + 1, 2^p]$. The height of the resulting tree is at most $\log |U|$. To insert a new element x, we compare x with the root of the tree, say, r. Let $2^{i-1} < r \leq 2^i$. We distinguish the following cases:

Case 1. $x > 2^i$: Make x the tree root with the old tree as the left subtree.

Case 2. $r < x \leq 2^i$: Insert x into the right subtree recursively.

Case 3. $2^{i-1} + 1 \leq x < r$: Place x at the tree root, substituting item r, and insert r into the right subtree recursively.

Case 4. $x \leq 2^{i-1}$: Insert x into the left subtree recursively.

Deletion is analogous to insertion. It is easy to see that each operation requires only $O(\log |U|)$ rotations or CONS operations.

3. Set equality testing. In this section we describe a data structure for set equality testing with $O(\log m)$ amortized time and $O(\log m)$ space per update operation, where m denotes the total number of updates. The elements seen so far are numbered in serial order and define the current universe $U = [1, |U|]$. Each set is represented by a binary trie [11] in this universe. The binary trie representing a set S is an S-expression that stores the elements of S as atoms and is defined recursively. Let $2^p < |U| \leq 2^{p+1}$. A singleton set is represented by an atom, and the empty set is represented by the atom NIL. If $|S| \geq 2$, then S is represented by a pair $(s_1.s_2)$, where s_1 and s_2 are, respectively, the S-expressions representing subsets $S \cap [1, 2^p]$ and $S \cap [2^p + 1, |U|]$ in their respective subuniverses. We store S-expressions uniquely so that two sets are equal if and only if their S-expressions are represented by the same node. Unique storage of S-expressions makes CONS operations expensive. Fortunately, a set update operation translates into a cascade of at most $\log |U| \leq \log m$ CONS operations, which can be implemented in $O(\log m)$ amortized time and $O(\log m)$ space by using the method described in the following.

We now describe an efficient data structure for performing cascades of CONS operations on uniquely stored S-expressions. The data structure requires $O(f + \log m_c)$ amortized time to perform a cascade of f CONS operations, where m_c denotes the total number of cascades performed. Consider the collection of nodes representing S-expressions. Number the nodes serially in order of creation. Say node p is a *parent* of node v if p points to v. Each node, say, v, maintains a set *parents*(v) of all its parents. Each parent $p \in$ *parents*(v) is assigned a key equal to (serial#(w),b), where w is the other node (besides v) pointed to by p, and b equals 0 or 1 depending on whether the left pointer of p points to v or not. To perform a CONS operation on two nodes v and w, we search set *parents*(v) under key (serial#(w),0) and return the matching parent (or, alternatively, we search the set parents(w,0) under key (serial#(v),1)). If there is no matching parent, we create a new node p with pointers to v and w, set *parents*(p) to empty, insert p into *parents*(v) and *parents*(w), and return p. In a cascade of CONS operations we implement each CONS operation by searching in the set of parents of the node returned by the previous CONS operation.

We represent each set *parents*(v) by a binary search tree and use the splay algorithm [16] to perform searches. Insertion of a new item by using splay is implemented as follows. Maintain a pointer to the maximum item in the tree (in addition to the root pointer). If the inserted item is larger than the current maximum item, insert it as the right child of the maximum item. Otherwise, insert the item into the tree in the standard way and splay at the item. The two types of insertions are called *passive* and *active*, respectively. We implement passive insertions more efficiently since they are more numerous than active insertions.

The following theorem summarizes the performance of the data structure.

THEOREM 3. *The amortized cost of a cascade of f* CONS *operations is $O(f + \log m_c)$, where m_c is the total number of cascades performed on S-expressions.*

Before proving the theorem, we need a few lemmas. The following technical lemma uses the notion of blocks in a binary tree introduced in the proof of Theorem 1 and occurs implicitly in the work of Cole, Mishra, Schmidt, and Siegel [6].

LEMMA 2. *Consider a binary tree partitioned into blocks whose items have been assigned arbitrary nonnegative weights such that every block has positive weight (the weight of a block is the sum of the weights of items in the block). Let n denote the number of nodes in the tree, and let n_b denote the number of blocks. The cost of a sequence of m splays performed on the roots of the blocks is $O(m + n + \sum_{j=1}^{m} \log(W/w_j) + \sum_{i=1}^{n_b} \log(W/\bar{w}_i))$, where W is the total weight of all the items, w_j is the weight of the block of the jth accessed item, and $\bar{w}_i$ is the weight of the ith block of the tree.*

Proof. Assign potentials to all nodes in the tree as described by Cole, Mishra, Schmidt, and Siegel [6, §2], and analyze splays by using their analysis of global insertions. (An account of this analysis can also be found in Cole [5].) The amortized cost of splaying at the root of a block with weight w is $O(1 + \log(W/w))$, and the drop in potential over the entire sequence is $O(\sum_{i=1}^{n_b} \log(W/\bar{w}_i) + n)$. The result follows. □

The next lemma bounds the cost of the sequence of operations performed on a single parent set and is the key step in the analysis.

LEMMA 3. *Consider a sequence of insertions and searches performed on an (initially empty) binary search tree by using splays. Let f_i be the number of searches of item i, F be the total number of searches, n_a be the number of active insertions, and n be the total number of insertions. The cost of this sequence if $O(n + n_a \log n_a + F + \sum_{f_i \geq 1} f_i \log(F/f_i))$.*

Proof. We modify the sequence by preinserting all items into the initial tree according to their order of arrival (without splaying). On this tree we perform the searches and simulate the insertions. Active insertions are simulated by splaying at the corresponding items, and passive insertions are simply ignored. We obtain a sequence of splays corresponding to active insertions and searches (*active* splays and *hot* splays, respectively). It suffices to bound the cost of this sequence.

We bound the cost of this sequence by partitioning the tree into blocks and applying Lemma 2. Partition the tree into blocks as follows. Call the items accessed by active and hot splays *active* and *hot*, respectively. Every active or hot item forms a singleton block. Each nonempty interval of nodes between consecutive singleton blocks forms a *passive* block. Choose an item from each passive block, and call it the *block representative*. Note that n_a is the number of active items. The weight of item i is defined by

$$\left\{\begin{array}{ll} f_i & \text{if the item is hot,} \\ F/(n_a + 1) & \text{if the item is active but not hot,} \\ 0 & \text{if the item is in a passive block} \\ & \qquad \text{but not the representative.} \end{array}\right.$$

The representatives of $n_a + 1$ passive blocks are assigned a weight of $F/(n_a + 1)$ each, and the representatives of the remaining passive blocks are placed in one-to-one correspondence with the set of hot items and assigned the weights of their mates. The total weight of the tree is at most $4F$. If Lemma 2 is applied, the cost of the sequence of splays is

$$\begin{aligned} O\Bigg(& n_a + F + n + \sum_{f_i \geq 1} f_i \log\left(\frac{4F}{f_i}\right) + n_a \log(4(n_a + 1)) \\ & +2\left(\sum_{f_i \geq 1} \log\left(\frac{4F}{f_i}\right)\right) + (2n_a + 1)\log(4(n_a + 1))\Bigg) \\ = O & \left(n + n_a \log n_a + F + \sum_{f_i \geq 1} f_i \log\left(\frac{F}{f_i}\right)\right). \end{aligned}$$ □

Remark. The lemma is an extension of Sleator and Tarjan's static optimality theorem [16]. The theorem is restricted to sequences of searches in which all items of the tree are accessed at least once. Use of the analysis of Cole, Mishra, Schmidt, and Siegel [6] is what made our extension possible.

LEMMA 4. *Consider a digraph $G = (V, E)$, and consider a collection of walks in G. Let F_e be the number of traversals of edge e in the walks, F_v be $\sum_{(v,w)\in E} F_{(v,w)}$, for any vertex v, W_v be the number of walks originating at vertex v, and id_v be indegree$(v) + 1$. Then,*

$$\sum_{(v,w)\in E} F_{(v,w)} \log\left(\frac{F_v}{F_{(v,w)}}\right) \leq \sum_{v\in V} F_v \log id_v + \sum_{F_v \geq 1} W_v \log F_v.$$

Proof. Let $\bar{F}_{(v,w)}$ be $F_{(v,w)}$ minus the number of walks with (v, w) as the last edge. Then,

$$\begin{aligned}
\sum_{(v,w)\in E} F_{(v,w)} \log\left(\frac{F_v}{F_{(v,w)}}\right) &= \sum_{F_v\geq 1} F_v \log F_v + \sum_{(v,w)\in E} F_{(v,w)} \log\left(\frac{1}{F_{(v,w)}}\right) \\
&\leq \sum_{F_v\geq 1} W_v \log F_v + \sum_{(x,v)\in E} \bar{F}_{(x,v)} \log F_v \\
&\quad + \sum_{(v,w)\in E} \bar{F}_{(v,w)} \log\left(\frac{1}{F_{(v,w)}}\right) \\
&= \sum_{F_v\geq 1} W_v \log F_v + \sum_{F_w\geq 1} \sum_{(v,w)\in E} \bar{F}_{(v,w)} \log\left(\frac{F_w}{\bar{F}_{(v,w)}}\right) \\
&\leq \sum_{F_v\geq 1} W_v \log F_v + \sum_{w\in V} F_w \log id_w,
\end{aligned}$$

where the first inequality holds because $x \log(1/x)$ is decreasing in $[1/e, \infty]$ and the second follows from the entropy inequality. □

Proof of Theorem 3. Consider a sequence of m_c cascades of CONS operations comprising F CONS operations totally. The cost of a cascade of f CONS operations is $O(f)$ plus the cost of operations on parent sets. During any cascade of CONS operations, there are at most two active insertions into parent sets. These insertions are performed when the first node is created by the cascade, and all subsequent insertions are passive. Thus out of at most $2F$ insertions into parent sets during the entire sequence of cascades, at most $2m_c$ are active. Applying Lemma 3 to the sequence of insertions and searches performed on each parent set and summing the costs over all parent sets, we see that the total cost of parent set operations is

$$O\left(F + m_c \log m_c + \sum_{\text{nodes } v} \sum_{i\in parents(v)\wedge f_i\geq 1} f_i \log\left(\frac{F(v)}{f_i}\right)\right),$$

where $F(v)$ denotes the total number of searches performed on *parents*(v) and f_i denotes the number of searches of item i among these. The double summation bounds the total cost of all searches performed on the parent sets. We bound the summation, in turn, by using Lemma 4. The collection of nodes at the end of the sequence of cascades induces a directed graph with the nodes as vertices and edges from nodes to their parents. The indegree of each node in the graph is at most 2. For each edge (v, w) define $F_{(v,w)}$ to be the number of searches of node w performed on *parents*(v). Delete all edges e such that $F_e = 0$. If Lemma 4 is applied to the resulting graph, the summation is bounded by $F \log 3 + m_c \log m_c$. It follows that the cost of the sequence of cascades is $O(F + m_c \log m_c)$. The theorem follows. □

4. Sequence equality testing. In this section we describe a data structure for sequence equality testing with $O(\sqrt{n} \log m + \log m)$ amortized time and $O(\sqrt{n})$ amortized space per update, where n denotes the length of the sequence that is updated and m denotes the total number of updates. We represent a sequence by an almost complete binary tree analogous to the set representation described in §2.1. The ith node of the tree in symmetric order stores the ith element of the sequence. A j-run is a subsequence of $2^j - 1$ adjacent elements in the sequence. For each $j \leq \lfloor \log n/2 \rfloor$ we maintain a list of j-trees representing the j-runs of the sequence; this will allow us to update a sequence by creating $O(\sqrt{n})$ new trees through CONS operations. In this representation a node can belong to several sequences and hence can have left and right pointers corresponding to several lists of j-trees. This makes navigation through the list of j-trees of a particular sequence expensive. We make the data structure persistent so that it permits efficient navigation. A data structure is *fully persistent* [9] if it maintains all versions of the data structure created by the update operations so far and permits accesses and updates (that create new versions) to any existing version. Driscoll, Sarnak, Sleator, and Tarjan [9] show how to make a purely pointer-based data structure that has nodes of constant bounded indegree and constant bounded number of access points fully persistent by weakening the time and space per update operation to amortized bounds (the original bounds must be worst case). Their method is to split a node when it has to maintain too many fields corresponding to different versions of the data structure. In our data structure, nodes have indegree at most 4 since a node has at most two parents and at most two adjacent nodes in its list of j-trees that point to it. Also, the data structure is always accessed from the root of the binary tree. Therefore, it can be made fully persistent by using their technique. The persistent data structure maintains the representations of all sequences created so far and permits navigating in and updating any of them. The data structure supports an update operation on a sequence in $O(\sqrt{n})$ amortized time plus the time needed for $O(\sqrt{n})$ CONS operations.

We now discuss how to test equality of sequences by using this data structure. Two sequences are equal if and only if their binary trees are isomorphic; here we define *isomorphic trees* as trees having the same shape and the same labels on the nodes. To facilitate isomorphism testing of binary trees, we serially number each class of isomorphic binary trees in order of creation and store the serial number of a binary tree at its root. Two sequences are equal if and only if their binary trees have identical serial numbers. When a node of the persistent data structure is split, its serial number is stored in the resulting nodes. Let us see how to compute the serial number of a new tree created by a CONS operation. Number all elements seen so far in serial order, and store them in a balanced search tree so that the serial number of an element can be computed in $O(\log m)$ time. Associate with each tree a triple (l, r, x) of serial numbers of its two subtrees and root element. If the triple of the new tree equals the triple of some existing tree, its serial number is obtained from the serial number of the matching tree. Otherwise, the new tree forms a new isomorphism class, so it is assigned the next available serial number. We have reduced the computation of the serial number of a new tree to testing membership of its triple in the collection of triples of all the trees. The collection of triples associated with the trees is a subset of $[1, cm^{3/2}] \times [1, cm^{3/2}] \times [1, m]$ for some constant c. Willard [22] gives a data structure for maintaining a dictionary over an ordered universe U in $O(\sqrt{\log |U|})$ time and $O(1)$ amortized space per operation. If the collection of triples is maintained by using this data structure, the serial number of a new tree can be computed in $O(\sqrt{\log m})$ time and $O(1)$ amortized space. Willard's data structure requires a priori knowledge of the universe size. If the number of updates is not known a priori, we estimate this quantity tentatively and, whenever the actual number of updates overshoots the estimate, we square the estimate and reconstruct the data structure. This increases the amortized time per dictionary operation in Willard's data structure by only a constant factor.

The persistent data structure, in conjunction with the data structure for serial number computation, solves the sequence equality testing problem. The cost of testing equality of two sequences is $O(1)$. The amortized time to update a sequence of length n is $O(\sqrt{n \log m} + \log m)$; the amortized space required is $O(\sqrt{n})$.

5. Unique DAG representation of sequences. In this section we study unique representations of sequences by DAG's with nodes of bounded outdegree that are updated by using CONS operations. We show that any representation that represents all sequences of the same length by the same underlying DAG requires $\Omega(\sqrt{n}/\log n)$ CONS operations per update operation.

We describe our class of unique DAG representations. A *unique* DAG *representation* of sequences over a universe represents sequences by rooted, ordered DAGs with nodes of bounded outdegree. A DAG is a *rooted* DAG if all nodes of the DAG are reachable from a single node that is called the *root*. An *ordered* DAG is one in which the outgoing edges of any node are ordered. Let G be an ordered DAG, and let x be a node of the DAG with a sequence of outgoing edges $(x, y_1), (x, y_2), \ldots, (x, y_k)$. Node y_k is called the ith *outneighbor* of x. Represent a sequence of length n by an n-node rooted, ordered DAG whose nodes have outdegree at most c, and store the elements of the sequence, one element per node, in a way that is unique to the sequence; c is a fixed constant, independent of n. A unique DAG representation also has an additional property that makes it useful for equality testing: the representation represents distinct sequences by nonisomorphic DAG's; *nonisomorphic* DAG's are DAG's that either have different shapes or have different arrangements of elements in nodes. A collection of sequences is represented by a common DAG that contains the DAG's representing the individual sequences as its subgraphs. When a new sequence is created from an existing sequence by inserting or deleting an element, the DAG representing the new sequence is created by accessing some of its nodes from the common pool and creating the remaining nodes bottom-up from the accessed nodes. An update operation is also permitted to create any additional nodes that may be needed in the future. A CONS operation is used to create a new node given its element and its outneighbors. The cost of an update operation equals the number of CONS operations performed.

We seek a unique DAG representation of sequences that permits sequences to be updated in $O(\text{polylog}(n))$ CONS operations; n denotes the length of the updated sequence. Such a representation might lead to a polylogarithmic-time data structure for sequence equality testing. We study a restricted class of unique DAG representations, called *length-unique* DAG representations. A unique DAG representation is *length-unique* if it represents sequences of the same length by a fixed underlying DAG and it arranges the elements of these sequences in the nodes of the DAG according to a fixed permutation. Our main result in this section is the following theorem.

THEOREM 4. *Let n and $m \geq n$ be positive integers, and let U be a universe of size at least m. For any length-unique* DAG *representation of sequences over U, there is a sequence of m update operations on sequences of length at most n that causes $\Omega(m\sqrt{n}/\log n)$* CONS *operations to be performed.*

This lower bound says that the complete-binary-tree representation of sequences used in our sequence equality data structure is optimal, up to a log factor, among the class of length-unique DAG representations. The log factor in the denominator of the lower bound arises for a technical reason; we believe that the lower bound holds even without it. The lower bound can be extended to arbitrary unique DAG representations if the underlying universe is sufficiently large, with the help of Ramsey's theorem. This is not a direct consequence of Theorem 4, but it follows because the proof of the theorem also works for unique representations that represent sorted sequences of each length (not all sequences) over an ordered universe by a fixed underlying DAG, according to a fixed permutation.

The proof of the theorem uses two technical combinatorial lemmas. The first one concerns integer sequences. Define the *slope* of an integer sequence $a_1, a_2, \ldots, a_n$ to be $(a_n - a_1)/n$. Define the *sum* of two sequences $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ to be the sequence $a_1+b_1, \ldots, a_n+b_n$.

LEMMA 5. *Every sequence of n integers has a subsequence of length $n/(5\log n)$ that is the sum of an increasing sequence with slope at most $5\log n$ and a sequence of distinct elements.*

Proof of Lemma 5. The proof idea is that perturbing the sequence by a random linearly growing sequence gives a sequence with many distinct elements. Let $\bar{a} = (a_1, a_2, \ldots, a_n)$ be the sequence we want to decompose. Select a random number k from $\{1, 2, \ldots, n\}$. Define a sequence $\bar{r} = (r_1, \ldots, r_n)$ by $r_i = \lfloor ik/n \rfloor$. The sequence has slope at most 1. We claim that the sequence $\bar{d} = (a_1 - r_1, \ldots, a_n - r_n)$ is likely to have at least $n/(5 \log n)$ distinct elements. This would give the lemma.

Let us prove the preceding claim. Let p denote the number of distinct elements in sequence $\bar{d}$, and let $k_1, k_2, \ldots, k_p$ denote the respective numbers of occurrences of these p elements in $\bar{d}$. For a fixed pair of integers i, j such that $1 \le i < j \le n$ let us compute the probability of the event $d_i = d_j$.

$$\begin{aligned}
\Pr[d_i = d_j] &= \Pr[a_i - r_i = a_j - r_j] \\
&= \Pr[a_j - a_i = \lfloor jk/n \rfloor - \lfloor ik/n \rfloor] \\
&\le \Pr[a_j - a_i - 1 < (j-i)k/n < a_j - a_i + 1] \\
&= \Pr[n(a_j - a_i - 1)/(j-i) < k < n(a_j - a_i + 1)/(j-1)] \\
&\le 2/(j-i).
\end{aligned}$$

We estimate the expected value of $\sum_i k_i^2$.

$$\begin{aligned}
E\left[\sum_i k_i^2\right] &= 2E\left[\sum_i \binom{k_i}{2}\right] + n \\
&= 2\left(\sum_{i<j} \Pr[d_i = d_j]\right) + n \\
&\le 2\left(\sum_{i<j} \frac{2}{j-i}\right) + n \\
&\le 5n \log n.
\end{aligned}$$

We estimate $E[p]$ as follows:

$$\sum_i k_i^2 \ge \left(\sum_i k_i\right)^2 / p = n^2/p$$

by the Cauchy–Schwarz inequality. Hence,

$$E\left[\sum_i k_i^2\right] \ge E[n^2/p] \ge n^2/E[p]$$

by Jenson's inequality. It follows that $E[p]$ is at least $n/(5 \log n)$, completing the proof of the claim. The proof of the lemma is complete. □

The second technical lemma used in proving Theorem 4 concerns DAG's. Let G be a rooted, ordered DAG with node set $\{1, 2, \ldots n\}$. We say that two nodes v and w of G are *shape-isomorphic* if the subgraphs of G rooted at v and w are isomorphic as unlabeled, ordered diagraphs. We say that two nodes v and w are *rank-isomorphic* if the subgraphs of G rooted at v and w are shape-isomorphic and the shape-ismorphism between these subgraphs maps the ith largest labeled nodes in the subgraphs to each other, for all i.

LEMMA 6. *Let G be a rooted, ordered* DAG *with node set $\{1, 2, \ldots n\}$. There exist node triples $(u_1, v_1, w_1), (u_2, v_2, w_2), \ldots, (u_p, v_p, w_p)$ in G, where $p = \sqrt{n}/(12(c+1))$, and c is an upper bound on the outdegree of the nodes of G, such that all of the following hold*:

1. $|w_i - v_i| \ge \sqrt{n}$ *for all* i.
2. *Either* $v_i < w_i$ *for all* i *or vice versa.*

3. $|v_i - v_j| \geq \sqrt{n}$ *for all* $i \neq j$.

4. v_i *and* w_i *are descendants of* u_i *for all* i.

5. *If* u_i *and* u_j *are rank-isomorphic, then the isomorphism between the subgraphs rooted at these nodes matches* (v_i, w_i) *and* (v_j, w_j).

Proof of Lemma 6. We construct the desired node triples by using the following algorithm. We call a DAG node *narrow* if the labels of any two of its descendants differ by less than $2\sqrt{n}$; otherwise, we call the node *wide*. We call a DAG node *i-narrow* if it is the descendant of a narrow node that is the ith outneighbor of some wide node. Since every node is either wide or i-narrow for some i, it follows that either there are at least $n/(c+1)$ wide nodes or there are $n/(c+1)$ i-narrow nodes for some i. We consider the two cases separately:

Case 1. The number of wide nodes is at least $n/(c+1)$: We scan the wide nodes, one by one, marking them and simultaneously creating node triples (u_i, v_i, w_i) that satisfy Conditions 1, 3, 4, and part of Condition 5 of the lemma. Initially, none of the wide nodes is marked. Each scan step picks an unmarked wide node u, it adds a new triple (u, u, w), where w is the descendant of u whose label differs from u's label by the maximum amount, and it marks u as well as all wide nodes whose labels differ from u's label by less than $\sqrt{n}$. The scan continues until all wide nodes are marked. Since each scan step marks at most $2\sqrt{n} - 1$ wide nodes and there are at least $n/(c+1)$ wide nodes to be marked totally, the total number of scan steps as well as the total number of triples created is at least $\sqrt{n}/(2(c+1))$. The set of triples satisfies Conditions 1, 3, and 4 of the lemma. Instead of Condition 5, the triples satisfy the following weaker condition: For two triples (u_i, v_i, w_i) and (u_j, v_j, w_j) such that u_i and u_j are rank-isomorphic, the isomorphism between the subgraphs rooted at u_i and u_j always matches v_i and v_j but matches w_i and w_j only if $v_i - w_i$ and $v_j - w_j$ have the same sign. We can make the triples satisfy Conditions 2 and 5 by eliminating at most half of them.

Case 2. The number of i-narrow nodes is at least $n/(c+1)$ for some i: We create the triples by scanning the wide nodes in a way analogous to Case 1. Each scan step picks a wide node u whose ith outneighbor, say, v, is an unmarked narrow node, it creates a new triple (u, v, w) where w is the descendant of u whose label differs from v's label by the maximum amount, and it marks v as well as all i-narrow nodes whose labels differ from v's label by less than $\sqrt{n}$. When the scan terminates, the ith outneighbors of all wide nodes that are narrow nodes are marked. Hence at the end of the scan, for every i-narrow node, say, x, there is a marked i-narrow node whose label differs from x's label by less than $2\sqrt{n}$; in fact, there is a triple (u, v, w) such that x's label differs from v's label by at most $3\sqrt{n} - 2$. For any triple (u, v, w) there are at most $6\sqrt{n} - 3$ i-narrow nodes whose labels differ from v's label by at most $3\sqrt{n} - 2$. It follows that the total number of triples created is at least $\sqrt{n}/6(c+1))$. We eliminate at most half of these triples in order to make then satisfy Conditions 2 and 5 of the lemma.

This completes the description of the algorithm. The algorithm creates at least $\sqrt{n}/12(c+1))$ node triples that satisfy the conditions of the lemma. □

Let us prove Theorem 4. The proof involves extending the ideas used in the proof of the lower bound for unique binary-search-tree representations (Theorem 2) in a nonobvious way. We would like to motivate the proof, first, by mentioning some preliminary approaches that do not work. Following the proof of Theorem 2 we can try to devise an adversary operation sequence consisting only of substitutions of elements in the sequence and shift operations that shift the entire sequence to the left or right by one position. This approach does not work since there is a unique binary-tree representation of sequences that processes substitutions and shifts in $O(\log n)$ CONS operations. This representation represents a sequence by a complete binary tree and arranges the elements in the tree according to the bit-reversal permutation; that is, the ith element of the sequence is stored in the node with symmetric order number equal to the reverse of the binary representation of i. We leave the proof that the representation works as an exercise. Another class of adversary operation sequences to try are those consisting of substitutions and shifts of a fixed segment of the sequence. This approach also does not give the lower bound we are aiming at since there is a unique binary-tree representation that supports substitutions and shifts of any fixed segment of the sequence in $O(n^{1/3})$ CONS operations; we leave this representation as an exercise. Hence the only adversaries that have a chance of giving a $\sqrt{n}$ lower bound are ones that shift several different segments of the sequences in an intermixed fashion.

Proof of Theorem 4. We devise an adversary that performs a sequence of $O(n)$ operations involving sequences of length n and causes the performance of $\Omega(n\sqrt{n}/\log n)$ CONS operations. The theorem will follow immediately once we have such an adversary. The adversary partitions the sequence into about $\sqrt{n}$ adjacent blocks, each consisting of $\sqrt{n}$ adjacent positions, and performs an intermixed sequence of *global shift* operations that shift the entire sequence to the right by inserting a new element at the left end and deleting the rightmost element and *local shift* operations that shift an individual block to the left by inserting a new element at the right end and deleting the leftmost element of the block. The global and local shift operations will be so designed that a specific set of $\sqrt{n}/\log n$ nodes of the DAG representing the sequence will have to be re-created during most global shift operations.

We define a set of $\sqrt{n}/\log n$ nodes of the DAG representing the sequences that the adversary will cause to be re-created many times during the adversary operation sequence. Let G be the DAG representing sequences of length n. We label the nodes of G by the positions in a sequence of length n to which they correspond. From now on, we will refer to nodes of G by their labels (or positions in a sequence of length n). By Lemma 6 there is a set of node triples $(u_1, v_1, w_1), \ldots, (u_p, v_p, w_p)$, for some $p = \Omega(\sqrt{n})$, such that the five conditions of the lemma are satisfied. Without loss of generality, we can strengthen the second condition of the lemma and assume that $v_i < w_i$ for all i. We call each node triple (u_i, v_i, w_i) a *tripole*; v_i and w_i are called the *minus pole* and the *plus pole* of the tripole, respectively, and u_i is called its *head*. We delete all the tripoles with minus poles in the set $\{1, 2, \ldots, p\sqrt{n}/2\}$ and use only the remaining tripoles; the number of remaining tripoles is at least $p/2$. We renumber the remaining tripoles from 1 to a number $q \geq p/2$ in ascending order of their minus poles. Consider the sequence of lengths of the tripoles $\{w_i - v_i \mid i = 1, 2, \ldots, q\}$. By Lemma 5 this sequence has a subsequence of $w_{i_1} - v_{i_1}, \ldots, w_{i_r} - v_{i_r}$, where $r = q/(5\log q)$, that is the sum of a sequence of integers $0 \leq s_1 \leq \cdots \leq s_r \leq q$ and a sequence of distinct positive integers. We delete all tripoles whose indices are not in the set $\{i_1, i_2, \ldots, i_r\}$ and renumber the remaining tripoles from 1 to r in ascending order of their minus poles.

Our next task is to define the adversary operation sequence so that the nodes $u_1, u_2, \ldots, u_r$ will be re-created many times. The idea is to send many distinct pairs of sequence elements into nodes v_i and w_i of G.

Partition the positions of a sequence of length n into $p/2$ blocks $B_1, B_2, \ldots, B_{p/2}$ each consisting of $\sqrt{n}$ adjacent positions and a block $B_{p/2+1}$ consisting of the remaining positions $\{p\sqrt{n}/2+1, \ldots, n\}$. For a set of integers S, define $S+i$ to be the set $\{x+i \mid x \in S\}$. Suppose we color the positions in each of the blocks either red or blue; for each block, all the positions in the block get the same color. For a tripole (u, v, w) and a block B_i, for some $i \leq p/2$, we say that the tripole-block pair $((u, v, w), B_i)$ is *compatible* if the positions in B_i are all red positions and the positions in $(B_i + (w - v))$ are all blue positions. We seek a coloring of the blocks that has a large number of compatible tripole-block pairs. A random coloring of the blocks achieves this. So let us randomly color the positions in each of the blocks $B_1, \ldots, B_{p/2}$ either red or blue with probability 1/2, independent of the other blocks; block $B_{p/2+1}$ alone is always colored blue. The probability that a fixed tripole-block pair is compatible is at least

1/8. It follows that the expected number of compatible tripole-block pairs is at least $pr/16$. We conclude that there is a coloring of the blocks that has at least $pr/16 = \Omega(n/\log n)$ compatible tripole-block pairs.

The significance of the coloring of the blocks and of the notion of compatible tripole-block pairs is the following. The red blocks are the blocks that undergo local shift operations, and the blue blocks are the blocks that remain static during the adversary operation sequence. A red block gets repeatedly shifted to the right through global shift operations. When the block sweeps through a minus pole position v corresponding to a compatible tripole (u, v, w), the head node u of the tripole gets re-created $\sqrt{n}$ times. Since there are $\Omega(n/\log n)$ compatible tripole-block pairs, we will get $\Omega(n\sqrt{n}/\log n)$ total node creations during the adversary operation sequence.

We describe the adversary. Let us assume that the universe is totally ordered. The adversary initially creates a sorted sequence of n elements in the universe, partitions it into $p/2+1$ red/blue blocks as described previously, and repeats the following operation sequence n times:

1. *Local shift*: For each block of red positions that is immediately followed by a minus pole position of a tripole do the following. Let i be the index of the tripole (recall that the tripoles are numbered from 1 to r in ascending order of their minus poles). Perform $s_i - s_{i-1}$ left-shift operations on the red block; assume $s_0 = 0$. Each left-shift operation deletes the smallest element of the block and inserts a fresh largest element at the other end while maintaining the sortedness of the whole sequence; here a fresh element is an element that was never in the sequence before.

2. *Global shift*: Shift the entire sequence one position to the right by deleting the largest element and inserting a fresh smallest element at the other end. This operation also shifts the blocks and their coloring one position to the right; it does not assign a color to the new first position.

The total number of operations performed by the adversary is $O(n)$.

Let us analyze this adversary. We will show that for any compatible tripole-block pair $((u, v, w), B)$ and element x within block B, a new copy of node u is created when element x moves to position v in the sequence. It will follow that $\Omega(n\sqrt{n}/\log n)$ CONS operations are performed in creating new copies of the head nodes. Let x be an element of a block B, and let (u_i, v_i, w_i) and (u_j, v_j, w_j) be tripoles that are compatible with B. Let $G_i(u_i)$ $(G_j(u_j))$ be the subgraph rooted at node $u_i(u_j)$ when element x reaches position $v_i(v_j)$ of the sequence. We claim $G_i(u_i)$ and $G_j(u_j)$ are not identical, that is, either their shapes are different or they have different arrangements of elements in their nodes. Let us prove this claim. If the two subgraphs are not shape-isomorphic, there is nothing to prove. If they are shape-isomorphic alone, not rank-isomorphic, then since the elements of the sequence are sorted, the arrangements of elements in the two subgraphs must be different. The last case to handle is when $G_i(u_i)$ and $G_j(u_j)$ are rank-isomorphic. In this case the elements stored in positions w_i and w_j, respectively, at the times when x is at positions v_i and v_j, respectively, are different. This is ensured by the shift operations ($s_j - s_i$ in number) performed on B between these two times and by the positions w_i and w_j lying in blue blocks at these respective times. It follows that the subgraphs are not identical, proving the claim. Hence, for each compatible tripole-block pair $((u, v, w), B)$ and element x within B, a new copy of node u is created when element x moves to position v in the sequence. Some $\sqrt{n}$ elements that are ever in B reach position v of the sequence during the adversary operation sequence. Hence the total number of copies of the head nodes of the tripoles that are created during the adversary operation sequence is at least $pr\sqrt{n}/16 = \Omega(n\sqrt{n}/\log n)$. The theorem follows. □

6. Open problems.

Our work raises some interesting open problems:

1. Our proofs of the lower bounds for unique binary-search-tree representations use Ramsey's theorem; as a consequence, they require the universe to be extremely large relative to the dictionary ($|U|$ must be at least a tower of n 2's). If $|U|$ is only exponential in n, do the lower bounds still hold?

2. Is there a data structure for performing CONS operations on S-expressions in constant amortized time and constant amortized space, in general?

3. Prove (or disprove) that the problem of maintaining sets over a small, finite universe U under the complete repertoire of set operations (i,e., union, intersection, difference, etc.) has no *efficient* solution. An *efficient* solution is one that implements all set operations in $O(\text{polylog}(|U|))$ time.

4. Is there a data structure for sequence equality testing with $O(\text{polylog}(n))$ time and space per update operation, where n is the length of the updated sequence? If we are willing to use randomization and tolerate a probability of error in answering queries, then this can be achieved by representing sequences by using the standard signature method of taking the remainder modulo a randomly chosen prime. View each sequence as the base-m representation of a number, where m is the total number of update operations performed on the sequences. The resulting numbers are at most m^m each. Choose a random prime p in the range $[m^{3+c} \log m, 2m^{3+c} \log m]$, where c is a constant, and define the signature of a sequence s to be $s \bmod p$. Then, any particular set of m sequences will be assigned distinct signatures with probability at least $1 - 1/m^c$. In order to efficiently update the signature of a sequence we represent a sequence by the leaves of a balanced binary search tree (in left-to-right order) and store at each node the signature of the subsequence corresponding to its subtree. Leaving mundane details to the reader, we see that the resulting data structure requires only $O(\log m)$ time and space per update operation and errs during a sequence of operations with probability at most $1/m^c$. Alternatively, Sundar [19] gives an error-free randomized representation of sequences that supports joins/splits in $O(\log^2 n)$ expected time/space.

5. Is there a unique representation of sequences over $\{0, 1\}$ of DAG's by bounded outdegree that can be updated in $O(\text{polylog}(n))$ CONS operations, where n is the length of the updated sequence?

Acknowledgment. We thank Mike Fredman for valuable discussions.

REFERENCES

[1] J. ALLEN, *Anatomy of LISP*, McGraw-Hill, New York, 1978.

[2] A. ANDERSSON AND T. OTTMANN, *Faster uniquely represented dictionaries*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1991, pp. 642–649.

[3] C. R. ARAGON AND R. G. SEIDEL, *Randomized search trees*, in Proc. 30th Annual IEEE Symposium on Foundations of J. Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 540–545.

[4] J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[5] R. COLE, *On the dynamic finger conjecture for splay trees*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990, pp. 8–17.

[6] R. COLE, B. MISHRA, J. SCHMIDT, AND A. SIEGEL, *On the Dynamic Finger Conjecture for Splay Trees. Part* I: *Splay sorting log n-block sequences*, Courant Institute Tech. Report, September 1989.

[7] K. CULIK II AND D. WOOD, *A note on some tree similarity measures*, Inform. Process Lett., 15 (1982), pp. 39-42.

[8] M. DIETZFELBINGER, A. KARLIN, K. MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT, AND R. E. TARJAN, *Dynamic perfect hashing*: *Upper and lower bounds*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 524–531.

[9] J. R. DRISCOLL, N. SARNAK, D. D. SLEATOR, AND R. E. TARJAN, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.

[10] R. L. GRAHAM, B. L. ROTHSCHILD, AND J. SPENCER, *Ramsey Theory*, John Wiley, New York, 1980.

[11] D. E. KNUTH, *The Art of Computer Programming*, *Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[12] L. LOVÁSZ, *Combinatorial Problems and Exercises*, North-Holland, Amsterdam, 1979.

[13] J. I. MUNRO AND H. SUWANDA, *Implicit data structures for fast search and update*, J. Comput. System Sci., 21 (1980), pp. 236–250.

[14] W. PUGH, *Incremental Computation and the Incremental Evaluation of Functional Programs*, Ph.D. thesis, Cornell University, Ithaca, NY, 1988.

[15] W. PUGH AND T. TIETELBAUM, *Incremental computation via function caching*, in Proc. 16th Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1989, pp. 315–328.

[16] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[17] D. D. SLEATOR, R. E. TARJAN, AND W. P. THURSTON, *Rotation distance, triangulations, and hyperbolic geometry*, J. Amer. Math. Soc., 1 (1988), pp. 647–682.

[18] L. SNYDER, *On uniquely representable data structures*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1977, pp. 142–146.

[19] R. SUNDAR, *Maintaining sequences under equality-tests in polylogarithmic randomized time*, in Proc. 12th Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 1992, Lecture Notes in Computer Science, Springer-Verlag, Berlin, to appear.

[20] M. N. WEGMAN AND J. L. CARTER, *New hash functions and their use in authentication and set equality*, J. Comput. System Sci., 22 (1981), pp. 265–279.

[21] R. WILBER, *Lower bounds for accessing binary search trees with rotations*, SIAM J. Comput., 18 (1989), pp. 56–67.

[22] D. E. WILLARD, *New trie data structures which support very fast search operations*, J. Comput. System Sci., 28 (1984), pp. 379–394.

[23] D. YELLIN, *Representing sets with constant time equality testing*, IBM Tech. Report, April 1990; also in Proc. 1st Annual ACM–SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, New York, 1990, pp. 64–73.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 45–49, February 1994

A COMPLEXITY INDEX FOR SATISFIABILITY PROBLEMS*

E. BOROS[†], Y. CRAMA[‡], P. L. HAMMER[§], AND M. SAKS[¶]

Abstract. This paper associates a linear programming problem (LP) to any conjunctive normal form ϕ, and shows that the optimum value $Z(\phi)$ of this LP measures the complexity of the corresponding *SAT* (Boolean satisfiability) problem. More precisely, there is an algorithm for *SAT* that runs in polynomial time on the class of satisfiability problems satisfying $Z(\phi) \leq 1 + \frac{c \log n}{n}$ for a fixed constant c, where n is the number of variables. In contrast, for any fixed $\beta < 1$, *SAT* is still NP complete when restricted to the class of CNFs for which $Z(\phi) \leq 1 + (1/n^{\beta})$.

Key words. Boolean satisfiability, polynomial algorithms, NP completeness

AMS subject classifications. 90C09, 68Q15, 68T99

1. Introduction.

We consider Boolean formulae in *conjunctive normal form* (CNF), that is, formulae of the type

$$\phi(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m} \left(\bigvee_{i \in P_k} x_i \bigvee_{i \in N_k} \overline{x}_i \right), \tag{1}$$

where $x_1, \ldots, x_n$ are Boolean variables, $\overline{x}_1, \ldots, \overline{x}_n$ are their *complements*, and $P_1, \ldots, P_m$, $N_1, \ldots, N_m$ are subsets of $\{1, \ldots, n\}$ satisfying $P_k \bigcap N_k = \emptyset$ for $k \in [m] = \{1, \ldots, m\}$. Each of the terms $C_k = \bigvee_{i \in P_k} x_i \bigvee_{i \in N_k} \overline{x}_i$ for $k \in [m]$ is called a *clause* of ϕ, and the quantity $|P_k \bigcup N_k|$ is called the *degree* of the clause. The *degree* of ϕ is the maximum degree of its clauses. The *size* of ϕ, denoted $|\phi|$, is the sum of the degrees of its clauses. The *satisfiability problem* for ϕ is to determine whether or not ϕ is *satisfiable*, i.e., whether or not there is an assignment $(a_1, \ldots, a_n) \in \{0, 1\}^n$ to the variables such that

$$\phi(a_1, \ldots, a_n) = 1. \tag{2}$$

The satisfiability problem is well known to be NP complete, even when the input is restricted to the class of degree 3 CNFs [3]. It is solvable in polynomial (linear) time in $|\phi|$ when ϕ is restricted to belong to the class of degree 2, or *quadratic* CNFs [1], [3], [5], and when ϕ is restricted to the class of *Horn* CNFs, i.e., to CNFs that satisfy $|P_k| \leq 1$ for all $k \in [m]$ [4], [6], [7].

The *complexity index* of the CNF ϕ (or of (2)) is the optimal value $Z(\phi)$ of the linear programming problem

$$Z(\phi) = \min Z \tag{3}$$

*Received by the editors April 1, 1992; accepted for publication (in revised form) July 9, 1992.

[†] DIMACS and RUTCOR, Rutgers University, New Brunswick, New Jersey 08903. The research of this author was supported in part by National Science Foundation grant DMS 89-06870, Air Force Office of Scientific Research grants 89-0512 and 90-0008, and Office of Naval Research grant N0001492J1375.

[‡] Department of Quantitative Economics, University of Limburg, 6200 MD Maastricht, The Netherlands. The research of this author was supported in part by National Science Foundation grant DMS 89-06870, Air Force Office of Scientific Research grants 89-0512 and 90-0008, and Office of Naval Research grant N0001492J1375.

[§] RUTCOR, Rutgers University, New Brunswick, New Jersey 08903. The research of this author was supported in part by National Science Foundation grant DMS 89-06870, Air Force Office of Scientific Research grants 89-0512 and 90-0008, and Office of Naval Research grant N0001492J1375.

[¶] Department of Computer Science and Engineering, University of California at San Diego, La Jolla, California 92093, and Department of Mathematics, Rutgers University, New Brunswick, New Jersey 08903. The research of this author was supported in part by National Science Foundation grant CCR89-11388.

such that

$$(4)\qquad \sum_{i\in P_k}\alpha_i + \sum_{i\in N_k}(1-\alpha_i) \le Z \qquad (k = 1,\dots,m), \text{ and}$$

$$(5)\qquad 0 \le \alpha_i \le 1 \qquad (i = 1,\dots,n).$$

We refer to the above linear program as $LP(\phi)$. We call a CNF *q-Horn* if its complexity index is at most 1. It is easy to verify that both the quadratic and the Horn forms are in this class. Q-Horn formulae were introduced in [2].

The purpose of this paper is to show that the complexity index sharply delineates between classes of "easy" and "hard" satisfiability problems. In particular, we shall see that, for any fixed $\varepsilon > 0$, the class of satisfiability problems with complexity index smaller than $1+\varepsilon$ forms an NP-complete class. This is to be contrasted with the fact that the satisfiability problem for a q-Horn formula is polynomially solvable, as shown in [2] (see also §2). To express our results more precisely, let us introduce one more definition.

Given a function $f\colon \mathbf{N}_+ \to \mathbf{R}$, we define the problem $SAT(f)$ as follows:
Instance: a CNF formula $\phi(x_1,\dots,x_n)$ with $Z(\phi) \le f(n)$.
Question: is ϕ satisfiable?

We shall prove:

THEOREM 1.1. *$SAT(1+\frac{c\log n}{n})$ is polynomially solvable for all $c \in \mathbf{R}$.*

THEOREM 1.2. *$SAT(1+(1/n^{\beta}))$ is NP complete for all $\beta < 1$.*

Theorem 1.2 implies, in particular, that $SAT(1+\varepsilon)$ is NP complete for any fixed $\varepsilon > 0$. Theorems 1.1 and 1.2 follow from more general results that will be stated and proved in §§3 and 4.

2. Preliminaries. If ϕ is a Boolean formula on variable set X, and W is a subset of variables, we define the *reflection* of ϕ with respect to W to be the formula ϕ_W obtained from ϕ by complementing all occurrences of the variables in W. We say that ϕ and ψ are *congruent* if $\psi = \phi_W$ for some subset W. Trivially, two congruent formulae are either both satisfiable or not.

Suppose that ϕ is a Boolean formula and $T = (Z, \alpha_1,\dots,\alpha_n)$ is a solution to $LP(\phi)$. Let W be the set of variables x_j such that $\alpha_j < 1/2$. If $W = \emptyset$ we say that the solution $S(\phi)$ is *aligned*. Otherwise, if we define $\beta_j = 1-\alpha_j$ for $j \in W$ and $\beta_j = \alpha_j$ otherwise, then $S = (Z, \beta_1,\dots,\beta_n)$ is an aligned solution to $LP(\phi_W)$. We refer to the pair (ϕ_W, S) as the *aligned reflection* of (ϕ, T).

We now review some basic facts about q-Horn formulae (more details on this topic can be found in [2]). A *QHY-partition* for the CNF ϕ is a partition of the variables into three disjoint sets Q, H, and Y that satisfy the following conditions:

1. No clause contains more than two variables (complemented or uncomplemented) from Q.
2. Every clause contains at most one uncomplemented variable from H.
3. Every clause containing an uncomplemented variable from H contains no variables from Q.

If $Y = \emptyset$ then we say that (Q, H) is a *QH-partition.* It is easy to see that if ϕ has a QH-partition then ϕ is q-Horn, by setting $\alpha_j = 1/2$ for each $x_j \in Q$ and $\alpha_j = 1$ for each $x_j \in H$. More generally, ϕ is q-Horn if it is congruent to a formula that admits a QH-partition. Conversely, if ϕ is q-Horn, then ϕ is congruent to a formula ψ with aligned solution $S = (Z, \alpha_1,\dots,\alpha_n)$ with $Z \le 1$. Setting Q to be the set of variables with $\alpha_j = 1/2$ and H to be the set of variables with $\alpha_j > 1/2$, yields a QH-partition of ψ. Thus we have:

THEOREM 2.1 [2]. *A Boolean formula is q-Horn if and only if it is congruent to a formula that admits a QH-partition of its variables.*

In particular, this implies that quadratic, Horn, and so-called renamable or disguised Horn formulae [8] are q-Horn.

THEOREM 2.2 [2]. *Given a QH-partition of ϕ, there is an algorithm that tests satisfiability of ϕ in time $O(|\phi|)$.*

We describe this algorithm informally and leave the details of the linear time implementation to the reader.

Phase 1: While there is a clause involving exactly one variable, assign to this variable the 0-1 value required to satisfy the clause; if some clause becomes unsatisfiable due to such successive assignments, then stop : the CNF is not satisfiable.
Phase 2: Assign the value 0 to all remaining variables in H.
Phase 3: Solve the remaining (quadratic) satisfiability problem.

Finally, we note a corollary to Theorem 2.2.

COROLLARY 2.3. *Given a QHY-partition (Q, H, Y) of ϕ, there is an algorithm that tests satisfiability of ϕ in time $O(|\phi|2^{|Y|})$.*

Proof. Each of the $2^{|Y|}$ assignments A to the variables of Y determines a Boolean function ϕ_A on variable set $Q \bigcup H$. Trivially, ϕ is satisfiable if and only if at least one of the functions ϕ_A is satisfiable. It is easy to see that (Q, H) is a QH-partition of each of the ϕ_A, so each of these satisfiability problems can be solved in time $O(|\phi|)$ by Theorem 2.2. □

3. A simple satisfiability algorithm. In this section, we prove Theorem 1.1 by describing a simple algorithm for satisfiability, whose running time on any n-variable CNF ϕ is bounded above by $2^{6n\epsilon(\phi)}|\phi| + p(|\phi|)$, where $p(\cdot)$ is a polynomial function, and where

$$\epsilon(\phi) = \begin{cases} 0 & \text{if } Z(\phi) \leq 1, \\ Z(\phi) - 1 & \text{if } Z(\phi) > 1. \end{cases}$$

This implies that the algorithm runs in polynomial time on the set of SAT instances with $Z(\phi) \leq 1 + \frac{c \log n}{n}$.

The first step of the algorithm is to compute a solution $S = (Z, \alpha_1, \dots, \alpha_n)$ to $LP(\phi)$. By replacing the pair (ϕ, S) by its aligned reflection, we may assume that $\alpha_j \geq 1/2$ for each $j \in [n]$. The running time of this first step is bounded by a polynomial function of $|\phi|$.

The second part of the algorithm is provided by:

THEOREM 3.1. *There is a polynomial time algorithm A, which takes as input a CNF $\phi(x_1, \dots, x_n)$ and an aligned solution $S = (Z, \alpha_1, \dots, \alpha_n)$ to $LP(\phi)$ and produces a QHY-partition of the variables such that $|Y| < 6n\epsilon(\phi)$.*

The third and final part of the algorithm is now provided by Corollary 2.3: given a QHY-partition with $|Y| < 6n\epsilon(\phi)$, the satisfiability problem for ϕ can be solved in time $O(|\phi|2^{6n\epsilon(\phi)})$.

Proof of Theorem 3.1. If $Z \leq 1$ then ϕ is q-Horn and we are done. So suppose that $Z > 1$ and set $\epsilon = Z - 1$ $(= \epsilon(\phi))$. If J is an interval, let X_J denote the set of variables x_j for which $\alpha_j \in J$. Define $L = \frac{1-\epsilon}{2}$ and $R = \frac{2+2\epsilon}{3}$. We will need the following lemma.

LEMMA 3.2. *Let a and b be real numbers such that $(a, b) \subseteq (L, R)$ and $b - a > \epsilon$. Then $Q = X_{[1/2, a]}$, $H = X_{[b, 1]}$, and $Y = X_{(a, b)}$ is a QHY-partition of the variables.*

Proof. It suffices to check the three defining conditions of a QHY-partition.

To verify Condition 1, assume (for contradiction) that for some k, $P_k \cup N_k$ contains the indices, say u, v, and w, of three variables of Q. Then for this clause we obtain the following contradiction:

$$\sum_{i\in P_k}\alpha_i + \sum_{i\in N_k}(1-\alpha_i) \geq (1-\alpha_u)+(1-\alpha_v)+(1-\alpha_w) > 3(1-(R-\epsilon)) = 1+\epsilon.$$

To verify Condition 2, assume (for contradiction) that for some clause of index k there are two variables $x_u, x_v \in H$ such that $u \in P_k$ and $v \in P_k$. Then again we have a contradiction:

$$\sum_{i\in P_k}\alpha_i + \sum_{i\in N_k}(1-\alpha_i) \geq \alpha_u + \alpha_v \geq 2b > 2(L+\epsilon) = 1+\epsilon.$$

To verify Condition 3, assume (again for contradiction) that for a clause of index k, there are variables $x_u \in Q$ and $x_v \in H$, such that $u \in P_k \cup N_k$ and $v \in P_k$. Then for this clause, we obtain the contradiction:

$$\sum_{i\in P_k}\alpha_i + \sum_{i\in N_k}(1-\alpha_i) \geq (1-\alpha_u)+\alpha_v \geq (1-a)+b > 1+\epsilon. \qquad \square$$

To complete the proof of Theorem 3.1, we use Lemma 3.2 to find a QHY-partition with Y sufficiently small. Select p to be the largest integer such that $\Delta = \frac{R-L}{p}$ is greater than ϵ. For $q \in \{0, 1, \ldots, p\}$, set $c_q = L + q\Delta$. For each $q \in [p]$, the pair c_{q-1}, c_q satisfies the hypotheses on a, b in Lemma 3.2. Thus it is possible to construct a QHY-partition with $Y = X_{(c_{q-1},c_q)}$, for any $q \in [p]$. Because the intervals (c_{q-1}, c_q) are disjoint, one of them satisfies $|X_{(c_{j-1},c_j)}| \leq n/p$. The definition of p implies that $p+1 \geq \frac{R-L}{\epsilon} > 1 + \frac{1}{6\epsilon}$, and hence $\frac{n}{p} \leq 6\epsilon n$.

It remains to observe that we must have $p \leq n$, which implies that the index j such that $|X_{(c_{j-1},c_j)}|$ is smallest can be found in linear time. This is true since if $p > n$ then one of the sets $X_{(c_{q-1},c_q)}$ is empty and there would exist a QHY-partition with $Y = \emptyset$, contradicting the fact that $Z(\phi) > 1$, i.e., that ϕ is not a q-Horn formula. $\square$

4. Proof of Theorem 1.2. Theorem 1.2 will follow easily from the following:

THEOREM 4.1. *Given a positive integer t and a CNF $\phi(x_1, \ldots, x_n)$ of degree d, $d \geq 3$, a CNF ψ on $(1+t)n$ variables such that*

(i) *ϕ is satisfiable if and only if ψ is satisfiable, and*

(ii) $Z(\psi) \leq 1 + (1-\frac{2}{d})\frac{1}{t}$,

can be constructed in time polynomial in t and $|\phi|$.

Proof. Let ϕ be given by (1). Let ψ be the CNF on variable set $\{y_{ij} | 1 \leq i \leq n, 0 \leq j \leq t\}$ whose clauses are:

$$\bigvee_{i\in P_k} y_{io} \bigvee_{i\in N_k} \bar{y}_{it} \qquad (k = 1, \ldots, m), \tag{6}$$

$$\bar{y}_{ij} \bigvee y_{i,j+1} \qquad (i = 1, \ldots, n;\ j = 0, \ldots, t-1), \text{ and} \tag{7}$$

$$y_{io} \bigvee \bar{y}_{it} \qquad (i = 1, \ldots, n). \tag{8}$$

Clauses (7) and (8) ensure that in any satisfying assignment of ψ, $y_{io} = y_{i1} = \ldots = y_{it}$ for each $i \in [n]$. Identifying $y_{io}, \ldots, y_{it}$ with x_i, one easily concludes that ϕ is satisfiable if and only if ψ is satisfiable.

To show that $Z(\psi) \leq 1 + (1-\frac{2}{d})\frac{1}{t}$, let us define

$$\alpha_{ij} = \frac{1}{d} + \left(1 - \frac{2}{d}\right)\frac{j}{t}, \qquad (i = 1, \ldots, n;\ j = 0, \ldots, t).$$

Consider now a clause of type (6). Since $\alpha_{io} = \frac{1}{d}$ and $\alpha_{it} = 1 - \frac{1}{d}$, we obtain

$$\sum_{i \in P_k} \alpha_{io} + \sum_{i \in N_k} (1 - \alpha_{it}) = |P_k \bigcup N_k| \frac{1}{d} \leq 1.$$

Similarly, for the clauses of type (8),

$$\alpha_{io} + (1 - \alpha_{it}) = \frac{2}{d} < 1.$$

Finally, for the clauses of type (7)

$$(1 - \alpha_{ij}) + \alpha_{i,j+1} = 1 + \left(1 - \frac{2}{d}\right) \frac{1}{t}. \qquad \square$$

Proof of Theorem 1.2. Recall that the satisfiability problem for CNFs of degree 3 (3-SAT) is NP complete. Let r be a positive integer. By Theorem 4.1, there is a polynomial time reduction that maps any CNF ϕ of degree 3 on p variables to a CNF ψ on $n = p^{r+1} + p$ variables with $Z(\psi) \leq (1 + 1/3p^r)$. For a fixed $\beta < 1$ we choose $r = \lceil \frac{\beta}{1-\beta} \rceil$, which implies

$$n^\beta = (p^{r+1} + p)^\beta \leq (2p^{r+1})^\beta < 3p^{(r+1)\beta} \leq 3p^r,$$

and therefore $Z(\psi) \leq 1 + (1/n^\beta)$. Thus any instance of 3-SAT can be reduced in polynomial time to an instance of SAT$(1 + (1/n^\beta))$, implying the theorem. $\square$

REFERENCES

[1] B. Aspvall, M. F. Plass, and R. E. Tarjan, *A linear-time algorithm for testing the truth of certain quantified boolean formula*, Inform. Process. Lett., 8 (1979), pp. 121–123.

[2] E. Boros, Y. Crama, and P. L. Hammer, *Polynomial-time inference of all valid implications for Horn and related formulae*, Ann. Math. Art. Intell., 1 (1990), pp. 21–32.

[3] S. A. Cook, *The complexity of theorem proving procedures*, Proc. Third Annual Symposium on Theory of Computing (1971), pp. 151–158.

[4] W. F. Dowling and J. H. Gallier, *Linear time algorithms for testing the satisfiability of propositional Horn formulae*, J. Logic Programming, 3 (1984), pp. 267–284.

[5] S. Even, A. Itai, and A. Shamir, *On the complexity of timetable and multicommodity flow problems*, SIAM J. Comput., 5 (1976), pp. 691–703.

[6] A. Itai and J. A. Makowsky, *Unification as a complexity measure for logic programming*, J. Logic Programming, 4 (1987), pp. 105–117.

[7] N. D. Jones and W.T. Laaser, *Complete problems for deterministic polynomial time*, Theoret. Comput. Sci., 3 (1977), pp. 105–117.

[8] H. R. Lewis, *Renaming a set of clauses as a Horn set*, J. Assoc. Comput. Mach., 25 (1978), pp. 134–135.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 50–70, February 1994

THE SET UNION PROBLEM WITH UNLIMITED BACKTRACKING*

ALBERTO APOSTOLICO[†], GIUSEPPE F. ITALIANO[‡], GIORGIO GAMBOSI[§], AND MAURIZIO TALAMO[¶]

Abstract. An extension of the disjoint set union problem is considered, where the extra primitive backtrack(i) can undo the last i unions not yet undone. Let n be the total number of elements in all the sets. A data structure is presented that supports each union and find in $O(\log n/\log\log n)$ worst-case time and each backtrack(i) in $O(1)$ worst-case time, irrespective of i. The total space required by the data structure is $O(n)$. A byproduct of this construction is a partially persistent data structure for the standard set union problem, capable of supporting union, find, and find-in-the-past operations, each in $O(\log n/\log\log n)$ worst-case time. All these upper bounds are tight for the class of separable-pointer algorithms as well as in the cell probe model of computation.

Key words. disjoint set union, deunion, unlimited backtrack, design and analysis of algorithms

AMS subject classification. 68C25

1. Introduction.

The disjoint *set union problem* has been studied extensively during the past two decades [1], [2], [5], [10], [19], [20], [22]. The problem consists of maintaining an efficient internal representation for a dynamic partition of an n-element set S that undergoes a sequence of operations of the following kinds:

- union(A, B, C): combine the two subsets of S named, respectively, A and B into a new set named C.
- find(x): return the name of the unique subset of S that currently contains the element x.

Initially, the partition of S consists of the n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$, and the name of set $\{i\}$ is i. Various conventions can be made about the way in which the name C is chosen in a union, and they give rise to a small number of variations of the problem. Typically, the name of every set at any time is maintained to coincide with the name (an integer in $[1, n]$) of one of the elements of that set. Also, the name C in a union is usually one of the names of the two input sets. Along these lines, C can be rigidly identified with A, or it can be left unspecified and result in either A or B at runtime, depending on the details of the implementation of a union. All such classes of restrictions do not affect the substance of the set union problem, but they allow the withdrawal of the third argument C from the format of a union. Throughout this paper we reason in terms of the primitive union(A, B), which combines the two subsets named A and B into a new set named either A or B.

The most efficient algorithms for the set union problem were devised by Tarjan and van Leeuwen [19], [22]. Such algorithms run in $O(n+m\alpha(m+n, n))$ time on a sequence consisting of at most $n-1$ unions and m finds. Here α is a functional inverse of the Ackermann function. No better performance is possible for the class of *separable-pointer algorithms* [20], [22], i.e., in the *pointer-machine* [12], [18], [20] model of computation. The storage of a pointer machine consists of an unbounded collection of records connected by pointers. Each record can contain an arbitrary amount of additional information, but no arithmetic is allowed in the computation of the address of a record. Separable-pointer algorithms must obey the following rules [2], [20]:

*Received by the editors November 13, 1989; accepted for publication (in revised form) July 29, 1992.

[†] Department of Computer Science, Purdue University, West Lafayette, IN 47907, and Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy. The research of this author was supported in part by the French and Italian Ministries of Education, by National Science Foundation grants CCR-89-00305 and CCR-92-01078, by National Institutes of Health Library of Medicine grant R01 LM05118, by U.S. Air Force Office of Scientific Research grant 90-0107, and by British Research Council grant SERC-E76797.

[‡] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Roma, Italy. Present address, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.

[§] Dipartimento di Matematica Pura e Applicata, Università de L'Aquila, L'Aquila, Italy.

[¶] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Roma, Italy.

(i) The operations must be performed on line.

(ii) Each set element is a node of the data structure. There can also be additional nodes.

(iii) (*Separability*) After each operation the data structure can be partitioned into subgraphs such that each subgraph corresponds exactly to a current set. No edge leads from a subgraph to another.

(iv) To perform find(x) the algorithm obtains the node v containing x and follows paths starting from v until it reaches the node that contains the name of the corresponding set.

(v) During any operation the algorithm may insert or delete any number of edges. The only restriction is that rule (iii) must hold after each operation.

Recently, Fredman and Saks [5], [6] showed that even in the powerful *cell-probe* model of computation, which encompasses the power of a random-access machine, no better performance than $O(n + m\alpha(m + n, n))$ is possible for a sequence of n unions and m finds.

Despite the low amortized [21] bounds, Blum [2] showed that the worst-case bound per operation for the set union problem is $O(\log n / \log\log n)$.[1] Also, this upper bound is known to be tight for the class of separable-pointer algorithms [2] and in the cell-probe model of computation [6].

In recent years, some variants of the set union problem have been considered, where individual unions or sequence of unions can be backtracked upon [7], [8], [13], [15], [24]. Such extensions are motivated by problems arising in the memory management by Prolog interpreters [9], [13], [14], [23], in the incremental execution of logic programs [15], and in the implementation of search heuristics for resolution [11], [17]. Along these lines, Mannila and Ukkonen [13] proposed the *set union problem with backtracking*, where a third operation, *deunion*, which undoes the last union not yet undone, is introduced.

Westbrook and Tarjan [24] proved that any separable-pointer algorithm for the set union problem with backtracking requires $\Omega(m \log n / \log\log n)$ time in performing a sequence of m find, union, and deunion operations. They also gave several algorithms with $O(\log n / \log\log n)$ amortized running time, thus matching this lower bound. The overall space required by these algorithms is $O(n)$ [24].

An extension of the set union problem with backtracking is considered in [7], [8]. In this extension a real number is assigned to each union as the *weight* of that union, and it is possible to backtrack either to the union of maximal weight or to a generic union performed in the past. This extension has both a *static* [7] and a *dynamic* [8] version, depending on whether or not the weights can be dynamically updated. Both versions can be solved in $O(\log n)$ worst-case time per operation and in $O(n)$ overall space [8]. Also, this upper bound is tight for the class known as *non-separable-pointer algorithms* [16].

In this paper we consider a generalization of set union with backtracking where, in addition to the usual union and find operations, a primitive backtrack(i) that undoes the last i unions not yet undone is introduced. We call this problem the *set union problem with unlimited backtracking*. An efficient solution to this problem is desirable in several applications, notably, in the implementation of search heuristics for Prolog interpreters [9], [11], [17], [23]. In that framework, sequences of unions correspond to unifications between terms [14], and a multiple deunion would enable one to quickly recover from an unsuccessful search by returning to one of the most promising states among those examined so far.

Since backtrack(1) is simply a deunion operation, the algorithms in [24] can be easily adapted to handle unlimited backtracks, within the same amortized time and space performance. If, however, backtrack(i) is regarded as a single operation, then such an implementation requires $\Omega(n)$ time in the worst case.

[1] Throughout this paper all logarithms are taken to the base 2, unless explicitly noted otherwise.

Our implementation of the set union problem with unlimited backtracking takes worst-case time $O(\log n/\log\log n)$ for each union or find operation and constant time for each backtrack(i), irrespective of i. We use $O(n)$ overall space. Clearly, the $\Omega(\log n/\log\log n)$ per-operation lower bounds of [2] and [6] still hold for our problem, so that our bound is tight both in the separable-pointer and cell-probe models of computation.

A byproduct of our construction is a partially persistent [4] data structure that supports each union, find, and find-in-the-past operation in $O(\log n/\log\log n)$ worst-case time, with $O(n)$ space usage. This is faster than the bound achieved in [15], but the specifications of a union used in [15] are slightly different.

2. Union-find with deunion. As mentioned in [24], the data structure proposed by Blum [2] could be easily adapted to support deunions in $O(\log n/\log\log n)$ time per operation and in overall space $O(n)$. In this section we carry out the details of this extension that will serve as a basis for the subsequent developments. We start by recalling the tree structure used in [2].

For any $k \geq 2$ a k-UF *tree* is a rooted tree T such that

(i) the root has at least two children;
(ii) each internal node has at least k children;
(iii) all leaves are at the same level.

Clearly, the height of a k-UF tree with n leaves is bounded by $\lceil\log_k n\rceil$. We say that a node of a k-UF tree is *slim* if it has fewer than k children and is *fat* otherwise. A consequence of the aforementioned definitions is that only the root of a k-UF tree can be slim. Disjoint sets are represented by k-UF trees as follows. The elements of the set are stored in the leaves, and the name of the set is stored in the root. Furthermore, the root also contains the height of the tree and a bit specifying whether it is fat or slim.

A find(x) is performed by first climbing up the tree from the leaf containing x and then returning the name stored in the root. This takes $O(\log_k n)$ time.

To perform union(A, B) for two nonsingleton sets A and B, we need access to the roots r_A and r_B of the corresponding k-UF trees T_A and T_B. Blum assumed that his algorithm obtained r_A and r_B in constant time, prior to performing a union(A, B). If this is not possible, r_A and r_B can be obtained by means of two finds (i.e., find(A) and find(B)), because of the property that the name of a set is one of the elements of that set. Once r_A and r_B are available, the two k-UF trees T_A and T_B are combined as follows.

Assume without loss of generality that height(T_B) $\leq$ height(T_A). Let v be the node on the path from the rightmost leaf of T_A to r_A such that the subtree of T_A rooted at v has the same height as T_B. Node v is found by starting at r_A and then following the leftmost downward branch of each node for exactly height(T_A)$-$ height(T_B) steps. After node v is reached, the manipulations to be performed depend on the type of union, according to the following.

Type 1 — Root r_B is fat (i.e., has no fewer than k children), and v is not the root of T_A. Then r_B is made a sibling of v.

Type 2 — Roots r_A and r_B are fat, and v is the root of T_A. A new (slim) root r is created and both r_A and r_B are made children of r.

Type 3 — This type covers all remaining possibilities. Specifically, if root r_B is slim, then the children of r_B are made the rightmost children of v. If root r_B is fat, then, since we are not in type 1 or 2, we have that $v = r_A$ and v is slim. In this case all the children of r_A are made rightmost children of r_B.

Note that new arcs are created only as part of a type 1 or 2 union. Type 3 unions involve instead what we call *redirecting* existing arcs. We make the assumption that the node representing a singleton set is a fat node. From now on, we fix $k = \lceil\log n/\log\log n\rceil$. This choice of k is motivated by the following theorem by Blum [2].

THEOREM 2.1 [2]. UF *trees support either union or find in* $O(\log n/\log\log n)$ *time and* $O(n)$ *space.*

Blum [2] proved also that this bound is tight for the class of separable-pointer algorithms. Recently, this result was extended to the cell-probe model of computation by Fredman and Saks [6].

THEOREM 2.2 [2], [6]. *Any separable-pointer algorithm for the disjoint set union problem has single-operation worst-case time complexity* $\Omega(\log n/\log\log n)$. *The same lower bound holds in the cell-probe model of computation.*

A UF tree can be easily adapted to support deunions. We list the few upgrades needed. The resulting structure will be called a DUF *tree*. For each node v the children of v are also linearly ordered from left to right in a doubly linked list. Two DUF trees T_A and T_B are combined in much the same way as UF trees, except that type 3 unions are now expanded as follows. Assume root r_B is slim. All the children of r_B are made the rightmost children of v. The arc connecting the leftmost child of r_B to v is marked a *separator*, and the label of r_B (i.e., the old name of the set represented by T_B) is recorded in that arc. Similar manipulations are performed when r_A is slim.

Because of the linear order on the children of each node, each union can be implicitly described by its *characteristic arc*, defined as follows. The characteristic arc of a type 1 union is $(r_B, \text{parent}(v))$. The characteristic arc of a type 2 union is (r_A, r). Finally, the characteristic arc of a type 3 union is the separator associated with that union. With the help of a stack P, characteristic arcs allow quick deunions to be performed. Following each union, a pointer to its characteristic arc is pushed onto P, along with the type identifier (1, 2, or 3) of that union. Type 1 and type 2 unions are then easily undone in constant time, following the pointer to the characteristic arc. To undo a type 3 union we access the separator pointed to by the top of the stack and disconnect this arc and all the arcs to its right. All the nodes so detached from the tree are made children of a new root to which the name stored in the separator is assigned. By the definition of type 3 union, this requires $O(k)$ time. Note that $O(n)$ nodes and arcs can be in the structure at any time. The stack records correspond to unions not yet undone, and there can be at most $n - 1$ such unions. Therefore, the total space required is $O(n)$. In conclusion, the following theorem holds.

THEOREM 2.3. DUF *trees support each union, find, and deunion in* $O(\log n/\log\log n)$ *time and* $O(n)$ *space.*

3. Upgrading DUF trees. In the set union problem with unlimited backtracking, deunions are replaced by backtracks: for any integer $i \geq 0$ backtrack(i) undoes the last i valid unions performed. Backtrack(i) is performed on DUF trees in $O(i\ \log n/\log\log n)$ time, simply by carrying out i deunions as described in §2. This is clearly undesirable, since i can be $\Theta(n)$. On the other hand, as long as we insist on deleting arcs the moment that they are invalidated by backtracking (i.e., in the *eager* mode [24]) the cost of backtrack(i) is $\Omega(i)$, since at least one arc must be removed for each erased union. To sidestep this lower bound, the removal of arcs invalidated by backtracking must be deferred to some possible future operation. This mode of operation is called *lazy*. In a strict sense, the lazy approach infringes on the separability condition stated in the introduction. However, the substance of that condition would still be met if one maintains that an arc is never traversed once it is invalidated (see, e.g., [24]). Our approach guarantees this fact and thus does not depart substantially from the separability assumption.

In what follows we present a data structure suitable for storing a collection of disjoint sets in such a way that the identity of each set in the collection is preserved. We call this data structure a k-BUF tree or, with the implicit assumption that $k = \lceil\log n/\log\log n\rceil$, simply a BUF tree. We will show that BUF trees support union and find in $O(\log n/\log\log n)$ time and backtrack(i) in constant time, independent of i.

We begin by describing the main features of BUF trees and by highlighting the associated implementation of the union, find, and backtrack operations. BUF trees retain the basic structure of DUF trees but differ from them primarily because of some implicit attributes defined on the arcs. With BUF trees a union is still performed according to one of three different patterns of management, as with DUF trees. In particular, we will have that type 1 and type 2 unions *create* new arcs, whereas type 3 unions only *redirect* already existing arcs. With BUF trees, however, a union must perform some additional manipulations on arcs, besides those pertaining to the mere aggregation of the two input subsets. In the following we say that an arc e is *handled* by a certain union only if e is either created or redirected by that union during the aggregation stage of that union. The main difference with DUF trees is that now, because of the lazy approach, we allow arcs and separators to possibly survive in the data structure also after the union that introduced them has been invalidated by backtracking. At any given time we call a union *valid* if it has not yet been undone by backtracks, and we call it *void* otherwise. We further partition void unions as follows. A void union is *persisting* if the arcs handled by that union have not yet been actually removed from the data structure, and it is *dissolved* otherwise. This classification of unions induces a corresponding taxonomy on arcs and separators as follows. In a BUF tree an ordinary arc can be *live*, *dead*, or *cheating*, and a separator arc can be, in addition, either *active* or *inactive*. Informally, live arcs represent connections not yet invalidated by backtracks; this happens when the last union that handled them is still valid. Dead arcs instead represent connections that, although still in the structure, only await destruction; this happens when the first union that created them is a void persisting union. Between live and dead arcs lie cheating arcs. They occur when the first union that created them is valid but the last union that handled them is a persisting type 3 union. Therefore, they represent faulty connections that do not have to be destroyed but have only to be replaced by the corresponding correct connections. As in DUF trees, separators are associated with type 3 unions. At any given time a separator is *active* if its associated union is valid and is inactive otherwise. A node of a BUF tree is *live* if there is at least one live arc entering it and is *persisting* otherwise. In analogy with the nodes of DUF trees, the live nodes of BUF trees can be *slim* or *fat*, but this is decided based only on the number of live arcs entering each node. Specifically, a node is slim if the number of live arcs entering it is less than k and is fat if the number of live arcs entering it is at least k.

Assume that we perform an intermixed sequence σ of union, find, and backtrack operations starting from the initial partition of S into n singletons. The partition of S that results from σ is the same as that produced by applying to S, in the same order as in σ, only those unions that are valid (i.e., not undone by backtracks) at the completion of σ. The subsequence of σ consisting only of unions that are still valid by the end of σ (i.e., by neglecting the unions made void by backtracking) is called the *virtual sequence of unions*. The following rules ensure that at any time each currently valid union u is assigned a unique integer $\text{ord}(u)$ representing the ordinal number of u in the current virtual sequence of unions:

(i) The first union performed gets ordinal number 1.

(ii) When a union is made void by backtracking, it relinquishes its ordinal number.

(iii) A new union gets an ordinal number equal to 1 plus the ordinal number of the last valid union performed.

At some point of the execution of σ, let $i_{\max}$ be the ordinal number of the last valid union performed so far. Backtrack(i) consists of removing the effect of the last i valid unions, that is, the effect of the last i unions in the current virtual sequence of unions. We perform backtrack(i) simply by setting $i_{\max} = \max\{i_{\max} - i, 0\}$, i.e., in constant time irrespective of i. This implementation of backtrack does not affect any arc in the forest, but its effect is implicitly recorded in the change of status of some arcs and separators. Part or all of these arcs might be removed or redirected later, while subsequent union operations are performed. In any event, we need to ensure the consistency of the forest of trees under this newly introduced operation. By the forest being consistent we mean that each tree in the forest stores a collection of sets in the current partition in such a way that, for any x, a find(x) executed as specified in the following correctly returns the name of the set currently containing x. We refer to the consistency of the forest as Invariant 0. The complete specification of this invariant requires some additional notions.

First, each arc e in a BUF tree T has two unions associated with it, as follows. The first union, denoted first_ union(e), is the union that created e. The second union, last_ union(e), is the last union not yet actually undone (i.e., either a valid or a persisting union) that handled e. We will maintain that ord(first_ union(e)) $\leq$ ord(last_ union(e)) for every arc e. In a consistent BUF tree an arc e is *dead* if and only if first_union(e) is void (i.e., e has to be destroyed since it gives a connection made void by some intervening backtrack). Similarly, arc e is cheating if and only if first_union(e) is valid and last_union(e) is void (i.e., e gives a faulty connection and hence has to be replaced but not completely destroyed). Finally, e is *live* (i.e., it gives a connection not yet affected by backtracking) if and only if last_union(e) is still valid. In addition to first_union and to last_union, each separator s also has associated the type 3 union that made it a separator. In the following, such a union will be referred to as separate_ union(s). A separator s is active if and only if separate_union(s) is valid and is inactive otherwise.

To complete our description of a consistent BUF tree T, let $S_1, S_2, \ldots, S_p$ be the disjoint sets stored in T. We specify the mapping from the set of leaves of T to the set of names of $S_1, S_2, \ldots, S_p$. Let x be a leaf of T and also a member of the set S_q, $1 \leq q \leq p$. Let Y be the name of S_q. Ascend from x towards the root of T following live arcs until a node without an outgoing live arc is met. Call this node apex(x). In a consistent BUF tree an apex falls always in one of the following three classes.

1. *Live apex* — There is no arc leaving apex(x), i.e., apex(x) is the root r of T. We will maintain that the name Y of S_q is stored in r.

2. *Dead apex* — The arc leaving apex(x) is dead. We will maintain that the name of S_q is stored in apex(x).

3. *Cheating apex* — The arc e leaving apex(x) is cheating. In this case we will maintain that at least one inactive separator falls within $k - 1$ arcs to the left of e and that the name of S_q is stored in the rightmost such separator.

The preceding description explains how a find is performed on a BUF tree. Throughout the sequence of union, find, and backtrack operations we need to maintain the forest of BUF trees in such a way that any arbitrary find would give a consistent answer. We formalize this condition as Invariant 0.

INVARIANT 0 (*find consistency*). Prior to the execution of each operation of a sequence σ of operations and for every element x of S, the following holds. If apex(x) is either dead or live, then the name of the set containing x is stored in apex(x). If apex(x) is cheating, then the name of the set containing x is stored in the rightmost inactive separator to the left of apex(x), and such a separator falls within $k - 1$ arcs to the left of apex(x).

The following fact is an immediate consequence of Invariant 0.

FACT 3.1. BUF *trees support each find operation in time $O((k+h)t)$, where t is the time needed to test the status of an arc and h is the maximum length of an ascending path from a leaf x to its apex in the tree.*

In the following sections we show that it is possible to implement BUF trees in such a way that t is $O(1)$ and h is $O(\log_k n)$. This immediately yields the claimed $O(\log n/ \log\log n)$ time bound for each find.

We now examine what is involved in performing union operations. Let A and B be two different subsets of the partition of S, such that $A \neq B$. In the collection of BUF trees that represents this partition, let T_1 and T_2 be the trees storing, respectively, A and B. We remark that two disjoint sets can happen to be stored in the same tree, so that T_1 and T_2 may coincide even if $A \neq B$. The first task of union(A, B) consists of finding in T_1 and T_2 the roots of the smallest subtrees that store, respectively, A and B. These roots are located by performing two finds. The associated subtrees have to be detached from their host trees and then combined into a single tree. Once the two subtrees have been located and detached, their unification requires a treatment quite similar to that of the union procedure described for DUF trees in §2. The most delicate part of the process, however, is in the first stage. The correctness of the two initial finds depends on our ability to preserve Invariant 0 through each union, find, and backtrack. This is discussed in the next sections.

4. Dominance trees and the procedure Restore. As said, we follow the lazy approach of undoing unions made void by backtracks not immediately, but rather during the execution of subsequent unions. Within the claimed time bounds, however, a single union cannot undo *all* the currently persisting unions. On the other hand, this is also not strictly necessary. What is necessary for a union is to undo all the persisting unions that undermine its own consistent execution, along with the validity of Invariant 0 on the resulting forest of BUF trees. It turns out that such a reduced task can be performed within the claimed time and space bounds, at the expense of some additional bookkeeping.

Our technique consists of maintaining the edges in every BUF tree T grouped into clusters, a *cluster* being defined as a maximal set E of consecutive sibling arcs with the property that last_union is the same for all the elements of E. We will maintain that the size of any cluster is at most $k - 1$ at any given time. At any point in the computation, a cluster is persisting if the field *last* common to its arcs exceeds the current value of $i_{\max}$ (i.e., if the last_union of its arcs is void), and it is live otherwise. This section describes the structure of such clusters and then details the operation of a procedure, called Restore, that will carry out a recurrent subtask of our BUF-tree implementation of a union. In informal terms, the task of Restore is that of removing all dead arcs from the input cluster, and then partitioning the remaining arcs in a certain number of smaller, yet live clusters. We will see that any union involves at most a constant number of calls to Restore and that the cost of each such call is $O(\log n / \log\log n)$ time.

Before describing the structural properties of clusters, we need to make some additional assumptions on the structure of BUF trees. To each arc e, two integers first(e) and last(e) are assigned. They represent, respectively, the ordinal number given to first_union(e) and to last_union(e). Besides first(s) and last(s), each separator s contains the following additional information. An attribute separate(s) is the ordinal number given to separate_union(s). Furthermore, label(s) is the name destroyed by separate_union(s), and number(s) is the total number of arcs moved during the execution of separate_union(s). These latter arcs will be maintained to fall immediately to the right of s. Since separate_union(s) is a type 3 union, number(s)$< k$. By the definition of a cluster, all the edges in a cluster E have the same value of *last* field. We refer to last(E) as the value shared by the last fields of all the arcs in E.

For each node v, fat(v) is the ordinal number of the last union that made v a fat node, provided that the effects of that union have not been actually removed from the data structure (i.e., that union is not a dissolved union). If no such union exists, then fat(v) is undefined. According to this convention, a slim node that was once fat may have a defined fat number. In addition to Invariant 0, we will maintain the invariants given below.

INVARIANT 1 (the $i_{\max}$ invariant). At any time, the following properties hold. For every arc e in a BUF tree, arc e is dead if and only if $i_{\max} <$ first(e), is cheating if and only if first$(e) \leq i_{\max} <$ last(e), and is live if and only if $i_{\max} \geq$ last(e). If, in addition, e is a separator, then e is inactive if and only if $i_{\max} <$ separate(e) and e is active if and only if $i_{\max} \geq$ separate(e). For every node v in the tree such that fat(v) is defined, fat$(v) \leq i_{\max}$ if and only if v is fat.

Maintaining Invariant 1 enables us to test the status of an arc in constant time. One more important consequence of this invariant is that either all arcs in a cluster are live or none is. Let now e and f be two arcs in a cluster E. We write $e < f$ if e precedes f in the left-to-right order, and we denote by $|f - e|$ the number of consecutive arcs between e and f (including both). If s is a separator in E, then we say that s *dominates* f if and only if $s < f$ and $|f - s| \leq$ number(s). We maintain also the following invariant.

INVARIANT 2 (the nesting invariant). Let E be a cluster. If $|E| = 1$, then the only element of E is not a separator. Assume now $|E| > 1$. Then, if the leftmost arc of E is a separator, say, s, then separate$(s) =$ last(s) and number$(s) = |E|$. If the leftmost arc of E is not a separator, then $|E| = 2$, E is the leftmost one among its sibling clusters, and E contains no other separators. In general, if s' and s'' are any two separators in E and s' dominates s'', then s' also dominates any arc e dominated by s''.

The nested structure of a cluster E delimited by a left separator is described in detail with the aid of a rooted, ordered tree called the *dominance tree* $D(E)$ of E. The leaves of $D(E)$ in preorder correspond bijectively to the arcs of E (including separators) from left to right; the internal vertices of $D(E)$ correspond bijectively to the separators. Thus, given a simple arc e in E, there is only one leaf ℓ in $D(E)$ corresponding to e, while there is a leaf ℓ and also an internal vertex v in $D(E)$ in correspondence with each separator of $D(E)$. If s is such a separator, then ℓ is the leftmost leaf in the subtree of $D(E)$ rooted at v.

The main feature of $D(E)$ is the following. Let ℓ be the leaf of $D(E)$ that corresponds to arc $e \in E$. Then, the internal vertices on the path from ℓ to the root of $D(E)$ correspond to the separators that dominate e, in the same succession as such separators are met in E starting from e and scanning E from right to left (see Fig. 1). In the following we will not distinguish between an arc or separator of E and its corresponding vertex in $D(E)$ whenever our meaning is made clear by the context.

Besides representing the nestings of separators, dominance trees encapsulate some monotonicity properties that form the object of our next invariant. Specifically, each vertex v in a dominance tree $D(E)$ gets assigned an integer $r(v)$ $(1 \leq r(v) \leq n)$, with the following meaning. If v is a leaf, then $r(v) =$ first(v). If v is an internal vertex (hence it maps a separator), then $r(v) =$ separate(v). We now consider all the arcs entering a node in a BUF tree as partitioned into clusters, and we assign similar numbers, denoted by R, to such clusters. Specifically, if either E is the leftmost cluster or $|E| = 1$, then $R(E) =$ first(e), where e is the leftmost arc in E. Otherwise, $R(E)$ is separate(s), where s is the separator that coincides with the leftmost arc in E (cf. Invariant 2). The numbers assigned in this way to the vertices of dominance trees and clusters of arcs entering a node will satisfy the monotonicity condition given below.

INVARIANT 3 (the monotonicity invariant). Let the dominance tree $D(E)$ of cluster E be defined. Hence $|E| > 1$. Then the two leftmost children of each internal vertex of $D(E)$ are always two leaves. Moreover, every internal node s (which must correspond to a separator) has number(s) equal to the number of leaves of the subtree of $D(E)$ rooted at s. Furthermore, if v and v' are sibling vertices of $D(E)$ with $v < v'$, then one of the following two cases applies: (i) if v and v' are the two leftmost vertices among their siblings (and thus leaves by the preceding part of this invariant) with $v < v'$, then $r(v) = r(v')$; (ii) otherwise, $r(v) < r(v')$. We also have that if p is the parent of v, then $r(p) > r(v)$. The individual clusters entering a slim node of a BUF tree obey the following rules. The leftmost cluster of arcs entering a slim node

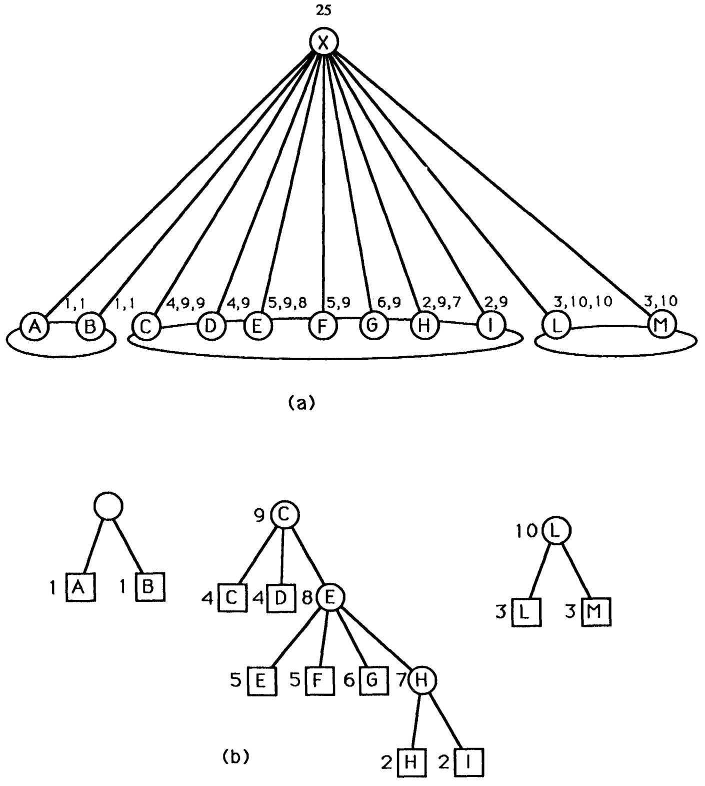


FIG. 1. BUF *trees and dominance trees.* (a) *The* BUF *tree produced for* $k > 10$, *by applying to an initial singleton partition the ordered sequence of unions*: union(A, B), union(H, I), union(L, M), union(C, D), union(E, F), union(F, G), union(F, I), union(D, G), union(A, E), union(B, L). *The first two numbers at the bottom right of each arc represent, respectively, the* first *and* last *field for that arc. Separators are also labeled with a third number, representing their* separate *field. This sequence produces* 3 *clusters and the* 4 *separators* (C, X), (E, X), (H, X), *and* L, X). (b) *Dominance trees associated with the clusters of* (a). *The number on the left of node* v *represents* $r(v)$.

contains two arcs and no separators. Furthermore, if E and E' are clusters of arcs entering the same slim node and E is on the left of E', then $R(E) < R(E')$.

As one of the consequences of Invariant 3 we get that if f is an arc dominated by a separator s, then first(f) < separate(s). Our last two invariants are as follows.

INVARIANT 4 (the slim compression invariant). The live arcs entering any slim node are leftmost among their siblings and have nondecreasing last fields, from left to right. For fat nodes this property holds for all the arcs that were directed to that node while the node was slim, including the arcs that made the node fat.

The slim compression invariant enables us to decide in $O(k)$ time whether a node is slim or fat simply by examining the at most k leftmost arcs entering that node.

INVARIANT 5 (the numbering invariant). For any integer i, $1 \leq i \leq n - 1$, there are either at most two sibling arcs with first field equal to i or at most one arc with separate field equal to i. Moreover, there are at most $k - 1$ sibling arcs with last field equal to i, and such arcs are in a cluster. Let E be this cluster. If E contains only one arc e, then first(e) = last(e). If $|E| > 1$, then the first two arcs of E have the same first field, the second arc of E is not a separator, and the remaining arcs possibly existing in E have first fields different from that of the first two arcs. Moreover, if the leftmost arc of E is not a separator, then $|E| = 2$ and the first fields of its two arcs are equal to their last fields. Otherwise, each arc in E has last field strictly greater than first field, and the leftmost arc has separator field equal to i. Finally, given i we can access in constant time the arcs with first field equal to i or with separator field equal to i.

The numbering invariant guarantees that the size of each cluster is at most $k - 1$ and that no two distinct clusters can have arcs with identical last fields. The last part of the invariant implies that a singleton cluster or a cluster not delimited by a left separator cannot contain cheating arcs. Thus such types of clusters contain either live or dead arcs.

We are now ready to describe how a persisting cluster of m arcs is detached from its host BUF tree in $O(m)$ time, maintaining Invariants 0–5 on the resulting dismembered structure. This is accomplished by the procedure Restore, which takes as input some arc e and an integer value $i_{\max}$. The specific tasks of Restore are as follows:

1. to identify the cluster E containing e;

2. to delete the dead arcs possibly existing in E;

3. to redirect the cheating arcs possibly existing in E towards newly introduced roots, in such a way that, by letting F be the forest of trees into which T has been dismembered, the following hold: (3.1) F represents, through Invariant 0, precisely the same collection of subsets of S formerly represented by T, and (3.2) all nondead arcs of E become live arcs in F.

To analyze what is involved in a Restore(e, $i_{\max}$), let E be the cluster containing e. If e is already live, then, by the $i_{\max}$ invariant, so are all the other arcs in E, so that Restore does not need to do anything. Henceforth we assume last(e) $> i_{\max}$, i.e., e is either cheating or dead. Then, by the definition of cluster, there cannot be any live arc in E. To deal with the most general case, assume that $D(E)$ is defined (i.e., E has a left separator) and let ℓ be a leaf of $D(E)$. With reference to the BUF tree T containing E, let v be the node from which arc ℓ originates, and let T' be the subtree of T rooted at v. Assume that ℓ is a dead arc of E. By Invariant 0 any leaf of T' connected to v by a path consisting solely of live arcs belongs currently to the set whose name is stored in v. Thus Restore can accomplish its task just by deleting ℓ. Assume now that ℓ is a cheating arc, and let as(ℓ) and is(ℓ) be, respectively, the highest active and lowest inactive separator on the path from ℓ to the root of $D(E)$. Observe that Invariants 2 and 5 guarantee that is(ℓ) is always defined in the case being considered. In the following, the expression "to the left of" is used to mean "to the left of and including."

LEMMA 4.1. *In E,* is(ℓ) *is the rightmost inactive separator to the left of ℓ.*

Proof. The assertion follows trivially from the definition of $D(E)$ if ℓ itself is an inactive separator in E. Thus we concentrate on the case where ℓ is not an inactive separator. Assume for a contradiction that the rightmost inactive separator to the left of ℓ in E is some s' such that $s' \neq$ is(ℓ) and $s' \neq \ell$ (see Fig. 2). By Invariant 2 and our choice of is(ℓ), separator s' cannot be on the path from ℓ to the root of $D(E)$ and thus does not dominate ℓ. Since s' falls in E between is(ℓ) and ℓ and since is(ℓ) dominates ℓ, then is(ℓ) dominates s', whence s' must lie in the subtree of $D(E)$ rooted at is(ℓ). Since s' is to the left of ℓ in E and $s' \neq \ell$, then in such a subtree of $D(E)$ we have that s' or an ancestor of s' is a left sibling of either ℓ or an ancestor of ℓ. Then let v stand for s' or the ancestor of s', and let v' stand for ℓ or the ancestor of ℓ, according to the case. By Invariant 3 and the $i_{\max}$ invariant, s' being an inactive separator implies that v is an inactive separator. By the same invariants, if v' is not ℓ, then v' is an active separator. If v' is an active separator, then always, by Invariant 3, $r(v) \leq r(v')$, whence v' active forces v to be active too, a contradiction. If v' coincides with ℓ, then the fact that ℓ is a cheating arc (i.e., first(ℓ) $\leq i_{\max}$), along with the conditions $r(v') =$ first(ℓ), $r(v) =$ separate(v), and $r(v) \leq r(v')$ (cf. Invariant 3), leads again to the contradiction that v is inactive. □

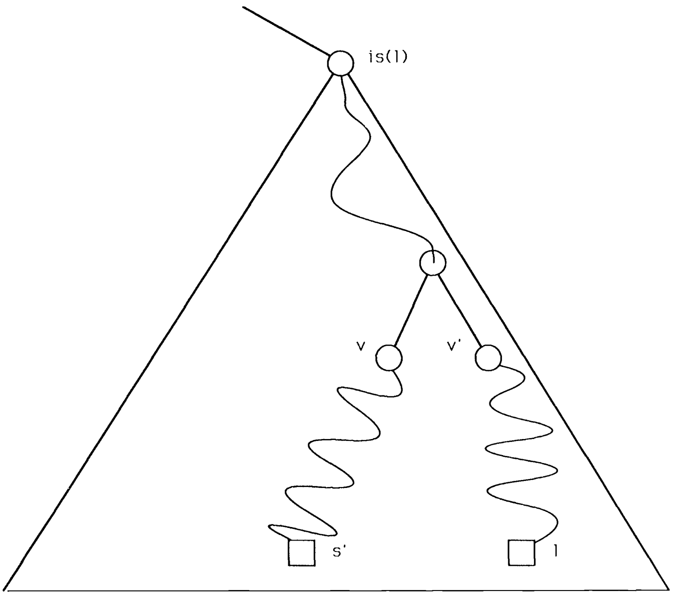


FIG. 2. *Illustration of Lemma* 4.1.

LEMMA 4.2. *If* as(ℓ) *is defined, then* as(ℓ) *is a direct son of* is(ℓ). *Moreover, the subtree of* $D(E)$ *rooted at* as(ℓ) *does not contain any inactive separator.*

Proof. If as(ℓ) and is(ℓ) are both defined, then from the fact that as(ℓ) is active and is(ℓ) is inactive we get that separate(as(ℓ)) < separate(is(ℓ)). By Invariant 3 is(ℓ) is then an ancestor of as(ℓ). That as(ℓ) is a son of is(ℓ) then follows straightforwardly from their respective definitions. The second part of the claim is an easy consequence of Invariant 3. □

Once $D(E)$ is given, it is easier to specify the operation of Restore so as to carry out tasks 1–3 consistently with Invariant 0. For this, let $E = \{e_1, e_2, \ldots, e_h\}, h < k$, be the cluster handled by Restore, and let x_i, $1 \leq i \leq h$, be the node of the BUF tree T from which arc e_i originates. As already observed, if one of the e_i's is live, then all the e_i's are live and Restore can terminate without affecting the structure of T. Assume therefore that E contains only cheating and dead arcs. The only leaves of T for which something must change are those whose previous apex was one of the x_i's. If x_i was a dead apex, then Restore will make x_i a live apex by simply deleting e_i. In this way the name of the set of leaves having apex in x_i remains the same. If x_i was cheating, then Restore will move the arc e_i to a new root with name label(is(e_i)) and will reset the last field of e_i to separate(as(e_i)). By Lemma 4.1, is(e_i) is the rightmost inactive separator to the left of e_i, so that also in this case all the nodes with apex x_i preserve their name. Each one of the aforementioned cases can be handled trivially in $O(k)$ time, but Restore must update *all* the arcs of E within this bound. The main handle for this is given by the nested structure of $D(E)$. To clarify this point, we describe a computation on $D(E)$ that we call Dismember (see Fig. 3).

The goal of Dismember is threefold. First, it will disconnect from its father every internal node of $D(E)$ that corresponds to an inactive separator. Thus in the forest of trees produced by Dismember no internal vertex other than a root can be an inactive separator. Second, Dismember will delete every leaf corresponding to a dead arc. Finally, Dismember will reset the last field of every surviving leaf ℓ to the separate field of the highest active separator on the path from ℓ to the root of $D(E)$ if such a separator exists and to first(ℓ) otherwise. Thus

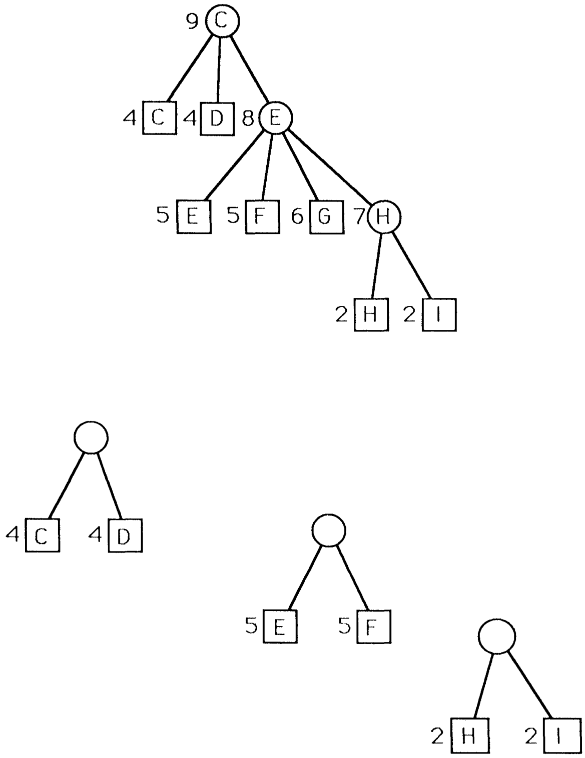


FIG. 3. *The effect of a dismember with* $i_{\max} = 5$ *on the second cluster of Fig.* 1(b).

there will be only live leaves in the output forest. Observe that these goals are unambiguous and mutually consistent, in force of Lemmas 4.1 and 4.2. The computation can be scheduled according to the preorder visit of the vertices of $D(E)$. It starts thus at the root of $D(E)$ and proceeds with the help of a stack P, which is used to store the inactive separators encountered in the visit. An inactive separator s is pushed onto P the first time it is visited and is popped from P immediately after all nodes in the subtree rooted at s have been handled. When a separator is popped from P, it is also disconnected from its father in $D(E)$ and it relinquishes its attributes as a separator in E. Assume that separator s was just pushed onto P. The computation considers all the children of s from left to right. If the child being considered is a leaf, then its last field is immediately updated. If it is an active separator, then Dismember visits the subtree of $D(E)$ rooted at such a separator updating all leaves in that subtree. Finally, if the child being considered is an inactive separator, then it is pushed onto P and the computation proceeds recursively on the children of such a separator.

It is clear that Dismember takes time $O(|E|)$. Assume that whenever Dismember deletes a leaf of $D(E)$ it also removes the corresponding arc of E. This accomplishes subtask 2 of Restore. The following few extra manipulations on the forest at the outset of Dismember suffice to accomplish subtask 3. First, for each tree D in that forest, the root x of a new BUF tree is created. Next, the arcs of E that are mapped into the leaves of D are considered in their left-to-right order, and each arc is redirected to x in succession, along with its applicable attribute fields (i.e., first and last and, for separators, also label, separate, and number). The only exception to this rule is represented by the leftmost arc, which corresponds to the leftmost leaf of D and also to the root of D. This arc surrenders its separator attributes, thus relinquishing its status as a separator, but its label field is stored into node x. The remaining separators are active, and they retain their attributes. Observe, incidentally, that the number field of each such separator is still consistent, in force of the second part of Lemma 4.2 (no pruning of $D(E)$ takes place below an active separator). At this point Lemma 4.1 and Invariant 0 yield that subtask 3.1 of Restore is accomplished provided only that every surviving arc of E is live. Recall that the only field changed by Dismember is the last field of cheating leaves and their associated separators. Specifically, the last field of a leaf ℓ is set equal to separate(as(ℓ)) if as(ℓ) is defined and is set equal to first(ℓ) otherwise. Invariant 3 guarantees then that every leaf of D has become live in this way, which accomplishes subtask 3.2.

We now consider subtask 1, and we also dispose of the cases where $D(E)$ is not defined. Clearly, Restore(e, $i_{\max}$) can check the status of e in constant time by the $i_{\max}$ invariant. For live e the procedure does nothing more. Thus we concentrate on the cases where e is either dead or cheating. By Invariant 5 and the definition of a cluster, the cluster E containing e is formed by at most $k - 1$ arcs. Thus E can be identified trivially in $O(k)$ time by checking the last fields of the arcs in an interval containing e and of size at most $k + 1$. If E is not delimited by a left separator, then (cf. Invariant 5 and the comment following it) we have $|E| = 2$, and the arcs in E are dead. The procedure deletes these two arcs and terminates, in constant time.

As is easily checked, there is no need to maintain dominance trees explicitly. The traversal of $D(E)$ performed by Dismember can be simulated by scanning E from left to right with an auxiliary stack. The stack is used as before to store the separators encountered in the scanning, a separator being kept in the stack until all the arcs within its dominion have been updated. Although $D(E)$ is not given explicitly, the procedure can use some easy bookkeeping on the number fields of the separators in order to detect the condition that the dominion of a separator has been exhausted. In conclusion, we can record the following theorem.

THEOREM 4.3. *There is an implementation of* Restore(e, $i_{\max}$) *that takes time* $O(k)$.

We now show that Restore preserves our invariants.

THEOREM 4.4. *The procedure* Restore *maintains Invariants* 0–5.

Proof. Let $E = \{e_1, e_2, \ldots, e_h\}, h < k$, be the cluster handled by Restore, and let x_i, $1 \leq i \leq h$, be the node of the BUF tree T from which arc e_i originates. Since Restore does nothing if one (hence every) e_i is live, we assume henceforth that E contains only cheating and dead arcs.

That Invariant 0 is maintained by Restore follows straightforwardly from the discussion preceding Theorem 4.3. That discussion also shows that Restore preserves the part of Invariant 1 that involves $i_{\max}$. Consider now the part of Invariant 1 that involves the fat field. Since all newly introduced nodes are slim by construction, these nodes do not have a defined fat field. The only other node of T whose fat field could be possibly affected by Restore is the node v that the arcs in the cluster E were entering prior to Restore. Since the arcs of E are not live, however, they did not contribute in any way to the fatness of v (only live arcs do). Since the procedure does not change the value of $i_{\max}$, then v will remain slim or fat after Restore, consistent with before.

To discuss the next invariants, consider the forest of trees produced by Dismember. We have already seen that every nontrivial tree in such a forest represents a collection of formerly cheating arcs of E that were changed into live arcs. We show now that Restore has actually done more than just resuscitate those arcs. Specifically, we claim that every nontrivial tree in the forest produced by Restore represents a collection of live clusters that obey, with their associated dominance trees, every applicable property in Invariants 2–5.

For this, let D be one of the trees produced by Dismember, and let s be the inactive separator of E that corresponds to the root of D. Consider the children of s in $D(E)$, from left to right. The first observation is that if s became the root of nontrivial tree D, then s has at least two children and the two leftmost children of s in D are precisely the two leftmost children of s in $D(E)$. In fact, let ℓ and ℓ' be the two leftmost children of s in $D(E)$. Then, Invariant 3 guarantees that first$(\ell) =$ first(ℓ'). If ℓ and ℓ' are both dead, then they are deleted. However, no sibling of ℓ in $D(E)$ could be a live leaf or an active separator in this case because of the monotonicity of the r-values prescribed by Invariant 3 for ℓ and its siblings. Hence s could not be the root of a tree in the forest built by Dismember. Assume now that ℓ and ℓ' are cheating. Then, as(ℓ) and as(ℓ') are not defined and is$(\ell) =$ is$(\ell') = s$. In this case s will be the root of a tree, within which ℓ and ℓ' will still be the two leftmost children of the root. Thus s has at least two children in D, and such children are leaves of D. These two leaves form the leftmost cluster in the new BUF tree created by Restore. By the horizontal monotonicity of Invariant 3, the size of this cluster is 2. By the operation of Restore, neither arc in the cluster is a separator. This cluster complies with every applicable part of Invariants 2–5.

The other children of the root s of D are either leaves or active separators that did not fall within the dominion of any other active separators of E. Let s' be one such child of s, and consider the two possible cases below.

Case 1: s' is a leaf. Then Dismember set last(s') = first(s'). Recall that, in $D(E)$, first(s') = $r(s')$. If s'' is the immediate right sibling of s', then $r(s'') > r(s')$ by Invariant 3. Hence s' becomes a singleton cluster in D, with R-number equal to the old r-number of s'.

Case 2: s' is an active separator of E. Recall that Dismember assigns to s' and all of its descendants a last field equal to separate(s') and leaves number(s') untouched. The subset of E that is represented by the leaves of D forms a cluster delimited by a left separator and with a consistent separator nesting. The R-number of such a cluster is the old r-number of s'. Clearly, the subtree of D rooted at s' is the consistent dominance tree of such a cluster.

In view of Lemmas 4.1 and 4.2, the preceding analysis shows that Invariant 2 is preserved by Restore. Since no number field or r-number is altered, then the part of Invariant 3 that concerns these fields and numbers is preserved. By the operation of Restore, the leaves in the subtrees rooted at the children of s will be directed towards the same root of a newly created BUF tree. Our analysis of cases 1 and 2 displays that the monotonicity of the r-numbers on the children of s before Dismember guarantees the monotonicity of the R-values of clusters entering this root. With regard to the node of the BUF tree that the arcs of E entered before Restore, clearly, the R-values of the former siblings of E were not affected, whence their relative order is preserved. Thus Invariants 2 and 3 are thoroughly maintained.

We now turn to Invariant 4. Since $|E| < k$ and all the redirected arcs will enter new nodes, no fat node is introduced by Restore. The novel slim nodes vacuously comply with Invariant 4 since all arcs entering them are live. The nondecreasing ordering of the last fields of such arcs is secured by Invariant 3. In fact, the new last fields are former r-numbers (i.e., either separate or first fields, depending on the type of node—an internal active separator or a leaf—encountered by Dismember), and these r-numbers obeyed Invariant 3. Consider now the BUF-tree node v that arcs in E entered before Restore. The only situation under which such arcs are disconnected from v is when they are not live. But in such a situation the arcs of E did not contribute in any way to the fatness or slimness of v. Thus Restore also preserves Invariant 4.

Finally, we deal with Invariant 5. Recall that Restore does not introduce new values for either first or separate fields. Furthermore, all the redirected arcs that get the same last field are siblings because of the implementation of Restore and do not exceed $k - 1$ since $|E| < k$ by hypothesis. We have seen that a new cluster E' created by Restore contains only one arc ℓ only if ℓ was a leaf in the input dominance tree $D(E)$ and as(ℓ) was not defined. In this case last(ℓ) was updated by Restore to first(ℓ), consistent with Invariant 5. A new cluster E', $|E'| > 1$, without a left separator is created by Restore only when the two leftmost leaves of $D(E)$ are encountered, and such leaves are assigned identical last and first fields. Otherwise, if $|E'| > 1$, E' was obtained as a subtree of $D(E)$ rooted at some active separator s. In this case, since Invariant 3 is already established, it follows that the two leftmost arcs in E' have the same first field and that the second arc in E' is not a separator. We have also seen that in this case all the arcs in E' get separate(s) as their new last field and that such a new last field is greater then all their first fields, also by Invariant 3. By definition of dominance tree, the leftmost arc in E' is the separator s. Thus for every leaf ℓ in E' we get that last(ℓ) = separate(s) as prescribed by Invariant 5. In conclusion, Invariant 5 is also preserved. □

Before we continue with our discussion it is instructive to revisit the outline of a BUF-tree union given at the end of §3. In that outline we said that a necessary preliminary stage of a union(A, B) consists of locating and detaching the roots of the two subtrees that contain A and B. But our description of Restore implies that in general the procedure also locates and detaches other trees that are not needed in the union. This is necessary in order to maintain a consistent record of the history encoded in the nested structure of clusters. Detaching only the subtrees of the BUF trees that are needed to perform the current union besets the consistency of the clusters that account for those subtrees at that moment. In particular, an edge e could be subtracted from the dominion of some separators without those separators becoming aware of this fact. This would infringe on the consistency of the number fields that are affected by the loss of e, thus undermining the consistency of future detachments.

5. Union-Find with unlimited backtracking. In this section we show that BUF trees support any union or find in $O(\log n / \log\log n)$ worst-case time and that they backtrack(i) in constant time, irrespective of i.

We study unions first. In terms of BUF trees, union(A, B) transforms the current input forest F of BUF trees into a new forest F' that meets the following specifications. First, F' represents, through Invariant 0, the same partition of S as F, except for the fact that A and B are consolidated into a single set. Second, Invariants 1–5 still hold on F'. Before proving this we describe how to support union(A, B).

To deal with the most general case, we assume that A and B are stored in two subtrees of some BUF tree(s) in F. The management of simpler cases is similar and will be omitted. Recall that union(A, B) must increment $i_{\max}$ by 1, the updated value of $i_{\max}$ being assigned to this union as its ordinal number. This increment of $i_{\max}$ may infringe on Invariant 5. To restore this invariant the procedure must remove from the forest F possibly existing arcs either with first field or separate field equal to $i_{\max}$. By Invariant 5 there were originally either at most two sibling arcs e' and e'' with first field equal to $i_{\max}$ or at most one arc e''' with separate field equal to $i_{\max}$, and such arcs can be accessed in constant time. The procedure deletes these arcs by means of either a Restore(e', $i_{\max}$) or a Restore(e''', $i_{\max}$), depending on the case. As a result, the forest F is transformed into an equivalent forest F'' no arc of which is labeled $i_{\max}$. By Theorems 4.3 and 4.4, F'' still satisfies Invariants 0–5, and F'' was produced in $O(k)$ time.

The next task consists of locating in F'', from input A and B, both apex(A) and apex(B). This stage is accomplished by performing two finds at a cost $O(k + h)$ (cf. Invariant 0) in the worst case, where h is the maximum possible length for a path originating at a leaf in a BUF tree and containing only live arcs. Clearly, Invariants 0–5 are not affected by this stage.

Next, union(A, B) transforms F'' into an equivalent forest F''' with the property that apex(A) and apex(B) are live in F'''. If we let e_A and e_B be the arcs, respectively leaving apex(A) and apex(B), this involves the two calls Restore(e_A, $i_{\max}$) and Restore(e_B, $i_{\max}$). Thus F''' is produced in $O(k)$ time, and it meets Invariants 0–5 because of Theorems 4.3 and 4.4.

Now let T_A and T_B be the BUF (sub)trees of F''' storing, respectively, A and B, and let r_A and r_B be their respective roots. The final task of union(A, B) is that of combining T_A and T_B into a single (sub)tree, thus producing the final forest F'. Assume without loss of generality that height$(T_B) \leq$ height(T_A). Observe that height(T_A) cannot exceed h since there is a live path from leaf A to r_A. Our BUF-tree union locates a live node v in T_A having the same height as r_B. This takes $O(h)$ steps, e.g., by retracking the find that produced r_A for height(T_B) steps. The procedure now selects one of the following three modes of operations, in analogy with a DUF-tree union.

Type 1 — r_B is fat and $v \neq r_A$. Root r_B is made a sibling of v, according to the following rule. If parent(v) is fat, r_B is made the rightmost child of parent(v). If parent(v) is slim, r_B is attached to the right of the rightmost live arc entering parent(v). At this point it is set first$((r_B, \text{parent}(v)))$ = last$((r_B, \text{parent}(v)))$ = $i_{\max}$. Finally, fat(parent(v)) is set to $i_{\max}$ if appropriate.

Type 2 — r_B and $v = r_A$ are both fat nodes. A new node r is created, and the name of r is copied from the name of either r_A or r_B. Next, both r_A and r_B are made children of r, thereby relinquishing their respective names. Finally, first$((r_A, r))$, first$((r_B, r))$, last$((r_A, r))$, and last$((r_B, r))$ are all set to $i_{\max}$.

Type 3 — This type covers all remaining possibilities, i.e., either root r_B is slim or root $v = r_A$ is slim. We describe only how the case of a slim r_B is handled, the other case being symmetric. By proceeding from left to right, every live child x of r_B is made a child of v, with the following policy. If v is fat, the newcomer arcs will be the rightmost arcs entering v. If v is slim, these arcs will be the rightmost live arcs entering v. The arc s connecting the leftmost child of r_B to v is marked a separator with separate$(s) = i_{\max}$. Moreover, the old name of r_B is stored into label(s), and number(s) is set to the total number of arcs moved. For every redirected arc e, last(e) is set to $i_{\max}$. Finally, fat(v) is set to $i_{\max}$ if appropriate.

To complete the management of union(A, B), a pointer indexed by $i_{\max}$ is directed towards the arc(s) (cf. type 1 or 2) or separator (type 3) introduced by the procedure. The fatness of a node can be tested in $O(k)$ time by a walk starting at its leftmost child (cf. Invariant 4).

We now prove the following lemma.

LEMMA 5.1. *Let h be the maximum possible length for a path originating at a leaf in a* BUF *tree and containing only live arcs. The preceding implementation of* union(A, B) *requires $O(k + h)$ time and preserves Invariants* 0–5.

Proof. The time bound is immediate from the preceding discussion. Therefore, we are left to show that a union preserves all the invariants. Since the three initial possible calls to Restore preserve the invariant, it is enough to show that the invariants are preserved after any of the three types of manipulations. Consider first type 1 and type 2 unions. They introduce new clusters with one and two arcs, respectively.

To see that find consistency is maintained, observe that only nodes in A and B may have their apex changed. If this change occurs, the new apex will be live and it will consistently store the name of $A \cup B$.

Consider now the $i_{\max}$ invariant. The only arcs for which something has changed are the newly introduced arcs: they are at most two, have last value equal to $i_{\max}$, and are live. Also, the last part of Invariant 1 still holds. For this, consider first a type 1 union, which introduces only the arc $(r_B, \text{parent}(v))$. The field fat(parent(v)) is unaffected if either parent(v) was fat or parent(v) was slim and did not become fat through the union. If, on the other hand, union(A, B) made parent(v) fat, then we have seen that it also sets fat(parent(v))= $i_{\max}$, thus preserving the last part of Invariant 1. In the case of a type 2 union, the new node r is introduced, and r is reached by the two arcs $((r_A, r)$ and $(r_B, r))$. Node r is thus slim, and fat(r) is, consistently, undefined. This completes the analysis of Invariant 1 for type 1 and type 2 unions.

Consider now the impact on Invariant 2 of type 1 and type 2 unions. A new cluster E with $|E| > 1$ can be created only by a type 2 union. In this case E does not contain any separator, and, having no siblings, it is vacuously the leftmost cluster. If E has only one arc, then E has been created by a type 1 union. Again, E contains no separators.

Invariants 3 and 4 are trivially maintained by a type 2 union, as well as by a type 1 union, the new arc introduced by which enters a fat parent(v). Consider now a type 1 union, the new arc introduced by which reaches a slim parent(v). In this case the new arc $e = (r_B,$ parent$(v))$ is inserted immediately after the rightmost live arc entering parent(v), and we have first$(e) =$ last$(e) = i_{\max}$. Thus first(e) will be larger than the first and separate fields of all of its left siblings, which consist only of live arcs or active separators in force of the slim compression and numbering invariants. Similarly, last(e) will be larger than the last field of every left sibling of e.

It is easily checked that a type 1 or type 2 union also maintains Invariant 5.

We turn now to type 3 unions. Let $e_i = (x_i, v)$, $1 \leq i \leq h < k$, be the arcs redirected by union(A, B) as they appear in the forest F'. Clearly, Invariant 0 is still valid in F'. In fact, the only nodes of F that had their apex changed are the nodes the old apex of which was one of the x_i's. The procedure provided for these nodes to have a new and consistent live apex.

The arcs $e_1, e_2, \ldots, e_h$ are live and have last field equal to $i_{\max}$. Moreover, separate$(e_1) = i_{\max}$, and e_1 is an active separator. The fat field of node v is correctly updated following a type 1 or type 3 union. Thus Invariant 1 is preserved.

A type 3 union introduces one new cluster $E = \{e_1, e_2, \ldots e_h\}$ by ordered aggregation of the clusters of edges entering a slim node of F'''. In the new cluster E the leftmost edge e_1 is made a separator. Furthermore, the last field of all the arcs in E will be set to $i_{\max}$, and therefore last$(E) =$ separate(e_1), as required by Invariant 2. Since Invariant 2 was valid in F''' for each one of the individual clusters contributing to E, number$(e_1) = |E|$. Thus E satisfies Invariant 2. By reasoning along the same lines, it is easy to check that E also satisfies Invariants 3 and 4.

The first part of Invariant 5 is preserved by the first call to Restore, while the rest of this invariant follows from the validity of Invariant 3 at the inception of the union. □

We now focus on the BUF-tree implementation of backtracks.

LEMMA 5.2. *For any values of* $i_{\max}$ *and* i*,* backtrack(i) *can be performed on a forest of* BUF *trees in constant time, preserving Invariants* 0–5.

Proof. As said, backtrack(i) is performed by setting $i_{\max} = \max(i_{\max} - i, 0)$, i.e., in constant time for any value of i. Hence we need to prove only that backtrack(i) maintains Invariants 0–5. Since the effect of a backtrack is null unless the value of $i_{\max}$ is altered, we can safely assume $i_{\max} - i \geq 0$. Then, we may regard a backtrack(i) as a sequence of i consecutive backtrack(1), and we need to prove only that if Invariants 0–5 were valid before performing a backtrack(1), they are still valid immediately afterwards. To fix the ideas, let $i_{\max}^{\text{old}}$ and $i_{\max}^{\text{new}} = i_{\max}^{\text{old}} - 1$ be the values of $i_{\max}$ immediately before and after backtrack(1), respectively.

We distinguish two cases, depending on the type of union undone. Let u be this union, and let A and B be the two sets unified by u.

If u is a type 1 or 2 union, then u introduced at most two arcs and such arc(s) are now made dead by the backtrack. Assume, for generality, that two arcs, say, $e_1 = (x_1, v)$ and $e_2(x_2, v)$, were introduced by u. Clearly, first(e_1) = first(e_2) = $i_{\max}^{\text{old}} > i_{\max}^{\text{old}} - 1 = i_{\max}^{\text{new}}$; hence these arcs become consistently dead. If e_1 and e_2 were cheating, then their death did not affect the fatness of v. If they were live, then v may have become, from fat, slim. But this implies that u was the last surviving union that made v fat; whence after backtrack fat$(v) = i_{\max}^{\text{old}}$ exceeds $i_{\max} = i_{\max}^{\text{new}}$. Since no arc other than e_1 and e_2 is affected by this backtrack, this guarantees the validity of Invariant 1.

Invariant 0 is also preserved. In fact, the only leaves the apex of which was possibly changed are those ending up with apex at either x_1 or x_2. The union operation u that the backtrack is voiding, however, did not delete the old names A and/or B stored in these nodes.

The leaves in the subtrees rooted at x_1 and/or x_2 are thus given back the old name A and/or B. This consistently reflects that u was made void.

The slim compression invariant is propagated by the validity, prior to backtrack(1), of Invariant 4 itself and of the part of Invariant 3 that concerns the R-numbers of clusters entering slim nodes. No part of Invariant 2, 3, or 5 is affected by a backtrack operation, so that these invariants are maintained too.

Assume now that u is of type 3, and let $e_1 = (x_1, v)$, $e_2 = (x_2, v), \dots, e_h = (x_h, v)$, with $h \leq k$, be the arcs issued by u. By hypothesis, the last field of these arcs was equal to $i_{\max}^{\text{old}}$ prior to backtrack and therefore is strictly larger than $i_{\max}^{\text{new}} = i_{\max}^{\text{old}} - 1$. Hence these arcs become consistently cheating (recall that, by Invariant 5, the first field of each arc in a cluster is strictly smaller than the last field of that arc). Since these arcs were live, v may become, from fat, slim. This means that u was the last surviving union that made v fat, and therefore $\text{fat}(v) = i_{\max}^{\text{old}}$ is now greater than $i_{\max}^{\text{new}}$. This settles Invariant 1.

Clearly, only the leaves that, prior to operation u, had apex at one of the x_i's are affected by the backtrack. By the structure of a type 3 union, however, the name of each such leaf was stored in label(e_1) as part of the execution of u. Having assumed Invariant 3 valid prior to the backtrack, we are guaranteed that, afterwards, e_1 is the rightmost inactive separator to the left of each e_i, $1 \leq i \leq h$. Thus Invariant 0 is preserved. Slim compression descends from the validity of Invariants 2 and 5 and of Invariant 4 itself prior to backtrack(1), while Invariants 2, 3, and 5 are all maintained vacuously. □

In order to prove our claimed time bounds, we show now that, at any time in a BUF-tree forest, the length of a path consisting of live arcs cannot exceed $O(\log_k n)$. This clearly establishes our bound for finds, and it combines with Lemma 5.1 to yield an identical bound for the union. Our desired property shall follow from the following lemma.

LEMMA 5.3. *At any time and for every arc* $e = (x, v)$ *in a* BUF-*tree forest, if* x *is not a leaf, then* $\text{fat}(x)$ *is defined and, moreover,* $\text{fat}(x) < \text{first}(e)$.

Proof. We proceed by induction on the number of operations performed. Initially there are n singleton trees and the claim holds vacuously since there are no arcs and no fat nodes in the structure.

Assuming now that the claim holds before the ith operation, $i \geq 1$, we prove that it also holds afterwards. The proof is straightforward in the case of finds and backtracks since these operations do not alter any of the parameters in the claim. Thus we concentrate on unions.

Then let the ith operation be union(A, B), where A and B are two arbitrary sets in the current partition of S.

We first show that the procedure Restore preserves the property of the claim. To see this, let $E = e_1, e_2, \dots, e_h$, $1 \leq h < k$, be the cluster of arcs managed by a Restore. As we know (see Invariant 1 and the definition of cluster), either all arcs in E are live or none is. Since Restore does nothing on live arcs, we concentrate on the case where E contains a mixture of dead and cheating arcs. We need to show that the claim holds after Restore for the arcs in E only, since every other arc or node was not affected.

Let x_i $(1 \leq i \leq h)$ be the node from which the arc e_i originated immediately prior to Restore. By the inductive hypothesis, $\text{fat}(x_i) < \text{first}(e_i)$ $(i = 1, 2, \dots, h)$. We now distinguish two cases for each arc e_i in E, depending on whether e_i is dead or cheating. If arc e_i is dead, then Restore simply deletes e_i, leaving a BUF tree rooted at x_i. The nodes of such a BUF tree still satisfy the invariant, by the inductive hypothesis. If, on the other hand, arc e_i is cheating, then by Invariant 1 $\text{first}(e_i) \leq i_{\max}$. By assumption $\text{fat}(x_i) < \text{first}(e_i)$, so that $\text{fat}(x_i) < i_{\max}$. As a consequence, the union that made x_i fat is still valid and therefore x_i is still fat. The procedure Restore redirects e_i to a new node, as explained in its description, which does not modify $\text{fat}(x_i)$ and $\text{first}(e_i)$. Since $\text{fat}(x_i)$ and $\text{first}(e_i)$ remain unchanged for each re-directed cheating arc e_i, the claim holds after Restore for every such arc. The nodes introduced by Restore are slim nodes and thus do not have a defined fat field. Clearly, no arc leaves such newly created nodes. In conclusion, Restore maintains our claim.

Recall that Restore is called for three times at the beginning of a union, the second and third time in order to produce the two trees T_A and T_B. We need now to show that the unification of T_A and T_B preserves the claim. Let r_A and r_B be the respective roots of T_A and T_B, and let $i_{\max}$ be the ordinal number of the present union. As usual, we distinguish three types of unions.

If a type 1 union is performed, then r_B is fat and therefore by Invariant 1 $\text{fat}(r_B) < i_{\max}$. A new arc e leaving from r_B is introduced, and $\text{first}(e)$ is set to $i_{\max}$. As a consequence, $\text{fat}(r_B) < \text{first}(e)$. Since this is the only change in the data structure, the claim is maintained.

If a type 2 union is performed, then r_A and r_B are both fat. Therefore, $\text{fat}(r_A) < i_{\max}$ and $\text{fat}(r_B) < i_{\max}$. The only change in the data structure is that a new node r and two new arcs (r_A, r) and (r_B, r) are introduced. Since $\text{first}(r_A, r) = \text{first}(r_B, r)$ is set to $i_{\max}$, then $\text{fat}(r_A) < \text{first}(r_A, r)$ and $\text{fat}(r_B) < \text{first}(r_B, r)$. Hence the claim is maintained.

If a type 3 union is performed, then either r_A is slim or r_B is slim. Assume to fix the ideas that r_B is slim. Then, at most $k - 1$ nodes (i.e., all the children x of r_B for which the arc (x, r_B) is live) are given a new parent v, but neither the first field of the redirected arcs (x, r_B) nor the fat field of the previous children of r_B is affected. As a consequence, we have only to check that the node v still fulfills the claim. If v was fat, then $\text{fat}(v) < i_{\max}$, and, by the inductive hypothesis, the arc leaving v (if any) had first field greater than $\text{fat}(v)$. If v was slim and there is no arc leaving v, then the claim will trivially still hold for v. Assume now that v is slim and there is an arc e leaving v. Then we claim that e must be dead. In fact, assume by contradiction that e is either live or cheating. This implies that, because of Invariant 1, $\text{first}(e) < i_{\max}$ and, because of the inductive assumption, $\text{fat}(v) < \text{first}(e) < i_{\max}$, which contradicts the hypothesis of v being slim. Therefore, the arc e leaving v must be dead. By Invariant 1 this is equivalent to saying that $\text{first}(e) > i_{\max}$. If the new children of v do not make it fat, then $\text{fat}(v)$ remains unchanged and the claim still trivially holds for v by propagation of the inductive hypothesis. On the other hand, if because of the type 3 union being performed v becomes fat, then $\text{fat}(v)$ changes to $i_{\max}$. But since e is dead, then $\text{first}(e) > i_{\max}$ and therefore $\text{fat}(v) < \text{first}(e)$. As a consequence, v will fulfill the claim in this case, too.

This completes the induction step of the union operation and establishes the lemma. □

Remark. The crucial implication of Lemma 5.3 is that live arcs can originate only from either leaves or fat nodes. Therefore, in any path composed only of live arcs, only the node at the top can be slim. Since, by definition, all arcs traversed by find(x) except the last one are live, it follows that a find(x) encounters only fat nodes on the path from x to apex(x), with the only possible exception of apex(x) itself. Then let $x = v_0, v_1, v_2, \ldots, v_{h-1}, v_h = \text{apex}(x)$ be the ordered sequence of nodes visited by a generic find(x) while climbing a path of length h up to apex(x). Then v_{h-1} has at least k live edges entering it, i.e., at least k fat children. Iterated application of Lemma 5.3 to these children, their own children, and so on yields that there are at least k^{h-1} leaves connected to v_{h-1} by means of paths consisting only of live edges. Since $k^{h-1} \leq n$, it follows that h is $O(\log_k n)$.

THEOREM 5.4. BUF *trees support each union and find in* $O(\log n / \log \log n)$ *time and support* backtrack(i) *in* $O(1)$ *time, irrespective of* i. *The overall space required is* $O(n)$.

Proof. The time bounds follow from Lemmas 5.1, 5.2, and 5.3. The space complexity of the data structure is dictated by the maximum number of arcs that may be present in it at any given time. New arcs are introduced only by unions, and each union can introduce at most two arcs. However, we have seen that when a union getting ordinal number i is performed, the arcs possibly created by a past union with the same ordinal number are removed from the data structure. This guarantees that at any time at most $2(n-1)$ arcs may exist in the data structure. If persisting nodes are removed as soon as there are no edges entering them, then the total space required by the data structure is $O(n)$. □

6. Conclusion. We have introduced a data structure for the efficient management of set union with unlimited backtracking. Our approach stays within the guidelines of separable-pointer algorithms if one only relaxes the separability condition to an extent that is deemed acceptable [24]. Our per-operation worst-case bounds are tight both for this model and for the more powerful cell-probe model of computation.

BUF trees also represent a partially persistent [4] data structure to be used in the following variant of the set union problem. In this variant the set union is defined as usual, but a find operation is formatted as find(x, k), where x is the name of an element of S and k is a nonnegative integer not exceeding the ordinal number of the last union so far performed. The task of find(x, k) is to return the name of the subset that contained the element x at the time when only the first k unions had been performed. To perform a find(x, k) on a BUF tree it is sufficient to temporarily set $i_{\max}$ to $i_{\max} - k$ and then proceed as per an ordinary find(x).

Acknowledgments. Moti Yung read an early version of this paper and gave several useful comments. A referee pointed out that a partially persistent set union algorithm for the RAM with logarithmic word size was recently given in [3].

REFERENCES

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

[2] N. BLUM, *On the single operation worst-case time complexity of the disjoint set union problem*, SIAM J. Comput. 15 (1986), pp. 1021–1024.

[3] P. DIETZ AND S. RAMAN, *Persistence and amortization*, in Proc. 2nd ACM–SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, New York, 1991, pp. 78-88.

[4] J. R. DRISCOLL, N. SARNAK, D. D. SLEATOR, AND R. E. TARJAN, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.

[5] M. L. FREDMAN, *On the Cell Probe Complexity of the Set Union Problem*, Tech. Report TM-ARH-013-570, Bell Communications Research, Red Bank, NJ, January 31, 1989.

[6] M. L. FREDMAN AND M. SAKS, *The cell probe complexity of dynamic data structures*, in Proc. 21st ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 345–354.

[7] G. GAMBOSI, G. F. ITALIANO, AND M. TALAMO, *Worst-case analysis of the set union problem with extended backtracking*, Theoret. Comput. Sci., 68 (1989), pp. 57–70.

[8] ———, *The set union problem with dynamic weighted backtracking*, BIT, 31 (1991), pp. 382–393.

[9] C. J. HOGGER, *Introduction to Logic Programming*, Academic Press, New York, 1984.

[10] J. E. HOPCROFT AND J. D. ULLMAN, *Set merging algorithms*, SIAM J. Comput. 2 (1973), pp. 294–303.

[11] T. IBARAKI, *M-depth search in branch and bound algorithms*, Internat. J. Comput. Inform. Sci., 7 (1978), pp. 313–373.

[12] D. E. KNUTH, *The Art of Computer Programming, Vol.* 1, *Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1968.

[13] H. MANNILA AND E. UKKONEN, *The set union problem with backtracking*, in Proc. 13th International Colloquium on Automata, Languages and Programming, European Association for Theoretical Computer Science, Rennes, France, 1986, pp. 236–243.

[14] ———, *Unifications, deunifications, and their complexity*, BIT, 30 (1990), pp. 599-619.

[15] ———, *Time parameter and arbitrary deunions in the set union problem*, in Proc. 1st Scandinavian Workshop on Algorithm Theory, Halmstad, Sweden, 1988, pp. 34–42.

[16] K. MEHLHORN, S. NAHER, AND H. ALT, *A lower bound for the complexity of the union-split-find problem*, SIAM J. Comput., 17 (1988), pp. 1093–1102.

[17] J. Pearl, *Heuristics*, Addison-Wesley, Reading, MA, 1984.

[18] A. Schönhage, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

[19] R. E. Tarjan, *Efficiency of a good but not linear set union algorithms*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[20] ———, *A class of algorithms which require non linear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[21] ———, *Amortized computational complexity*, SIAM J. Alg. Discrete Meth., 6 (1985), pp. 306–318.

[22] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[23] D. H. D. Warren and L. M. Pereira, *Prolog—the language and its implementation compared with LISP*, ACM SIGPLAN Notices 12 (1977), pp. 109–115.

[24] J. Westbrook and R. E. Tarjan, *Amortized analysis of algorithms for set union with backtracking*, SIAM J. Comput., 18 (1989), pp. 1–11.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 71–81, February 1994

006

THE PROFILE MINIMIZATION PROBLEM IN TREES*

DAVID KUO† AND GERARD J. CHANG†

Abstract. The profile minimization problem is to find a one-to-one function f from the vertex set $V(G)$ of a graph G to the set of all positive integers such that $\sum_{x \in V(G)} \{f(x) - \min_{y \in N[x]} f(y)\}$ is as small as possible, where $N[x] = \{x\} \cup \{y : y \text{ is adjacent to } x\}$ is the closed neighborhood of x in G. This paper gives an $O(n^{1.722})$ time algorithm for the problem in a tree of n vertices.

Key words. sparse matrix, profile, labeling, tree, leaf, centroid, basic path, algorithm

AMS subject classifications. 05C78, 05C85, 68R10

1. Introduction. The profile minimization problem was introduced by [5], [6] as a technique for handling sparse matrices. For instance, in the finite element method [8], [9], we want to solve a system of linear equations $Ax = b$ where A is a sparse symmetric $n \times n$ matrix. Suppose for each row i, $a_{ii} \neq 0$ and p_i is the position of the first non-zero element in this row. We call

$$w_i = i - p_i = i - \min\{j : a_{ij} \neq 0\}$$

the *width* of row i, and call

$$P(A) = \sum_{i=1}^{n} w_i$$

the *profile* of matrix A. To store A, we need only store $w_i + 1$ elements in each row i, which are from position p_i to position i. The total amount of storage for this scheme is then $P(A) + n$. In order to reduce the amount of storage, we need only permute the rows and columns of A simultaneously such that the resulting matrix has minimum profile, i.e., we need to find a permutation matrix Q such that the profile $P(QAQ^t)$ is minimized.

We can reformulate this problem in terms of graphs. Associate the matrix A with a graph G such that $V(G) = \{v_1, v_2, \ldots, v_n\}$ and $E(G) = \{(v_i, v_j) : i \neq j \text{ and } a_{ij} \neq 0\}$. Note that

$$P(A) = \sum_{i=1}^{n} w_i = \sum_{i=1}^{n} \left(i - \min_{v_j \in N[v_i]} j \right),$$

where $N[v_i] \equiv \{v_i\} \cup \{v_j : v_i \text{ is adjacent to } v_j\}$ is the *closed neighborhood* of v_i in G. The row and column permutation Q corresponds to a one-to-one function f from $V(G)$ onto $\{1, 2, \ldots, n\}$ and $P(QAQ^t) = \sum_{x \in V(G)} (f(x) - \min_{y \in N[x]} f(y))$. This motivates the definition of the profile of a graph given below.

For technical reasons, however, we shall give a slightly more general definition than that described in the previous paragraph. A *labeling* of a graph G is a one-to-one function f from the vertex set $V(G)$ to the set of all positive integers. A labeling is *simple* if it maps $V(G)$ onto $\{1, 2, \ldots, |V(G)|\}$. For a labeling f, the *profile-width* of a vertex x is defined as

$$w_f(x) = f(x) - \min_{y \in N[x]} f(y).$$

*Received by the editors March 11, 1991; accepted for publication (in revised form) July 30, 1992. This work was supported in part by the National Science Council of the Republic of China under grant NSC79-0208-M009-31.

†Institute of Applied Mathematics, National Chiao Tung University, Hsinchu 30050, Taiwan, Republic of China (gjchang@cc.nctu.edu.tw).

The *profile of G with respect to f* is

$$P_f(G) = \sum_{x \in V} w_f(x)$$

and the *profile* of G is

$$P(G) = \min\{P_f(G) : f \text{ is a labeling of } G\}.$$

A labeling f is *optimal* if $P_f(G) = P(G)$.

The purpose of this paper is to study the *profile minimization problem*, i.e., the problem of determining the profile $P(G)$ of a graph G, from an algorithmic point of view. The profile minimization problem is analogous to the *linear arrangement problem*, which is to find a labeling f of a graph G such that $\sum\{|f(x) - f(y)| : (x, y) \text{ is an edge in } G\}$ is minimized (see [1], [3], [7]). Reference [5] proved that the profile minimization problem is equivalent to the problem of interval graph completion, which is known to be NP-complete even when G is stipulated to be an edge graph (see [4]). The main result of this paper is to give an $O(n^{1.722})$ time algorithm for the problem when G is a tree of n vertices.

The rest of this paper is organized as follows. In §2, we establish several basic properties that motivate the development of our algorithm. In particular, we prove that for a tree T there exists a basic path $\alpha(x, y)$ such that $P(T) = P(T - \alpha(x, y)) + |E(T)|$. So the problem becomes that of finding a path $\alpha(x, y)$ such that $P(T - \alpha(x, y))$ is minimized. For the purposes of recurrence, we also introduce the problem of finding a path $\alpha(x, y)$ such that $P(T - \alpha(x, y))$ is minimized, with the boundary condition that y is fixed. In order to determine the basic path, §3 develops theorems that narrow the possibilities for the basic path. For instance, we prove that $\alpha(x, y)$ contains centroids of the tree. This also means that the number of vertices of each component of $T - \alpha(x, y)$ is no more than half the number of vertices of T. This is important in determining the speed of our recursive algorithm. Section 4 uses these results to design an algorithm, and §5 analyzes the time complexity of the algorithm.

2. Motivating properties. This section shows the existence of a basic path $\alpha(x, y)$ such that $P(T) = P(T - \alpha(x, y)) + |E(T)|$ and introduces the problem of finding a minimum such path with the boundary condition that y is fixed. The following properties are obvious and their proofs are omitted.

PROPOSITION 2.1. *An optimal labeling of a connected graph G maps $V(G)$ onto a set of consecutive integers.*

PROPOSITION 2.2 ([5]). *If H is a subgraph of G, then $P(H) \leq P(G)$.*

PROPOSITION 2.3 ([5]). *If G has m components $G_1, G_2, \ldots, G_m$, then $P(G) = \sum_{i=1}^{m} P(G_i)$.*

We can in fact assume that an optimal labeling of a graph is simple even if it is not connected. Suppose T is a tree of n vertices. For any leaf x and any vertex y in T, consider the unique (x, y)-path $\alpha(x, y) = (v_0, v_1, \ldots, v_r)$, where $v_0 = x$ and $v_r = y$. Suppose that for each i, $1 \leq i \leq r$, $T - \alpha(x, y)$ has n_i components $T_{i1}, T_{i2}, \ldots, T_{in_i}$ each with a vertex v_{ij} adjacent to v_i in T (see Fig. 2.1). Let f_i be an optimal simple labeling of $F_i = \cup_{1 \leq j \leq n_i} T_{ij}$. We consider a simple labeling f_{xy} defined by

$$f_{xy}(v) = \begin{cases} 1 & \text{if } v = v_0, \\ f_{xy}(v_{i-1}) + f_i(v) & \text{if } v \in V(F_i), \\ f_{xy}(v_{i-1}) + |V(F_i)| + 1 & \text{if } v = v_i. \end{cases}$$

See Fig. 2.2 for an example of f_{xy} with $\alpha(x, y) = (a, b, c, d)$. Note that the numbers beside the vertices are their labels. Then

$$w_{f_{xy}}(v_0) = 0,$$
$$w_{f_{xy}}(v_i) = f_{xy}(v_i) - f_{xy}(v_{i-1}) = |V(F_i)| + 1 \quad \text{for } 1 \le i \le r,$$

and

$$w_{f_{xy}}(v) = w_{f_i}(v) \quad \text{for } v \in V(F_i).$$

Consequently

$$\begin{aligned} P_{f_{xy}}(T) &= \sum_{i=1}^{r}(|V(F_i)| + 1 + P(F_i)) \\ &= |E(T)| + \sum_{i=1}^{r} P(F_i) \qquad (2.1) \\ &= |E(T)| + \sum_{i=1}^{r}\sum_{j=1}^{n_i} P(T_{ij}). \end{aligned}$$

We call $\alpha(x, y)$ the *basic path* (with respect to the labeling f_{xy}). Note that

$$1 = f_{xy}(v_0) < f_{xy}(v_1) < \cdots < f_{xy}(v_r) = n.$$

In general, an optimal labeling of a tree is of this type.

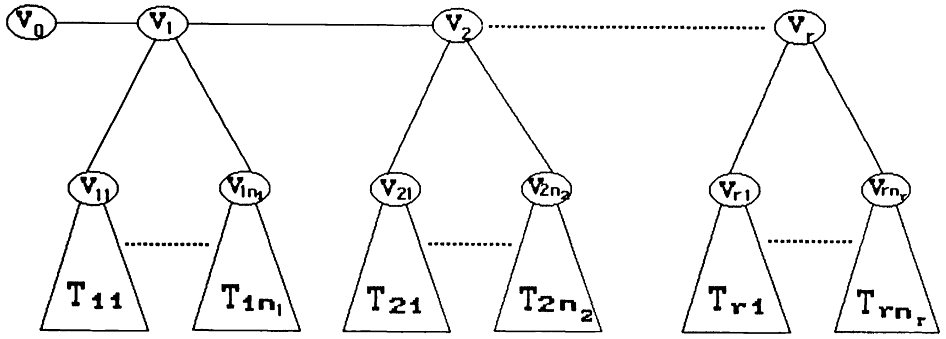


FIG. 2.1. *Tree T.*

THEOREM 2.4. *If f is an optimal labeling of a tree T of n vertices, then $f = f_{xy}$ where $x = f^{-1}(1)$ is a leaf and $y = f^{-1}(n)$ is adjacent to at most one non-leaf vertex.*

Proof. Let $\alpha(z, u) = (v_0, v_1, \ldots, v_r)$ be a longest path containing both x and y, say, $x = v_s$ and $y = v_t$ for $0 \le s < t \le r$. Note that since r is the maximum, v_0 and v_r are leaves. In this case $n_r = 0$ and $P_{f_{zu}}(T) = P_{f_{zu'}}(T)$, where $u' = v_{r-1}$.

Suppose T and $\alpha(z, u)$ are as shown in Fig. 2.1. Let $f_{ij} = f|_{V(T_{ij})}$ be the labeling f restricted on $V(T_{ij})$. Then, by definition,

$$P(T) = P_f(T) \ge \sum_{i=0}^{r} w_f(v_i) + \sum_{i=1}^{r}\sum_{j=1}^{n_i} P_{f_{ij}}(T_{ij}). \qquad (2.2)$$

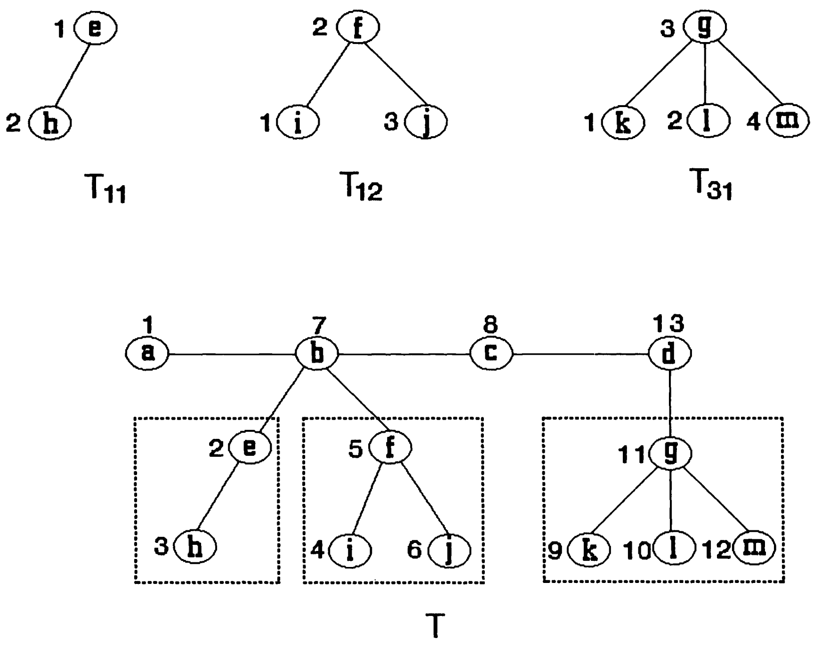


FIG. 2.2. *An example of f_{xy}.*

Note that

$$(2.3)\qquad \sum_{i=0}^{r} w_f(v_i) \ge \sum_{i=s+1}^{t} w_f(v_i) \ge \sum_{i=s+1}^{t} \{f(v_i) - f(v_{i-1})\} = n - 1 = |E(T)|.$$

Consequently, by (2.1),

$$(2.4)\qquad P(T) \ge |E(T)| + \sum_{i=1}^{r} \sum_{j=1}^{n_i} P(T_{ij}) = P_{f_{zu}}(T) \ge P(T).$$

Therefore, all inequalities in (2.2) to (2.4) are equalities. This implies the following:

(1) each f_{ij} is an optimal labeling for T_{ij},

(2) $w_f(v_0) = w_f(v_1) = \cdots = w_f(v_s) = 0$,

(3) $w_f(v_{t+1}) = w_f(v_{t+2}) = \cdots = w_f(v_r) = 0$,

(4) $f(v_{i-1}) = \min_{y \in N[v_i]} f(y)$ for $s + 1 \le i \le t$.

Statement (2) implies that $s = 0$, otherwise $w_f(v_{s-1}) = f(v_{s-1}) - f(v_s) > 0$. That is, $x = z$, which is a leaf. Statement (3) implies that $r-1 \le t$, otherwise either $f(v_{t+1}) > f(v_{t+2})$ or $f(v_{t+1}) < f(v_{t+2})$, i.e., either $w_f(v_{t+1}) > 0$ or $w_f(v_{t+2}) > 0$. So either $y = u$ or $y = u'$. In the former case, $y = u$ is a leaf. In the latter case, $y = u'$ is adjacent to at most one non-leaf vertex, otherwise we can choose a longer $\alpha(z, u)$. In this case, since $P_{f_{zu'}}(T) = P_{f_{zu}}(T)$, we replace u by u' and assume $y = u$. Note that in this case $n_r > 0$. So, now $\alpha(x, y) = \alpha(z, u)$. Finally, statement (4) implies the following:

(5) $1 = f(v_0) < f(v_1) < \cdots < f(v_r) = n$,

(6) $f(v_{i-1}) < f(v_{ij})$ for $1 \le i \le r$ and $1 \le j \le n_i$.

On the other hand, statement (1) and Proposition 2.1 imply that each $f(V(T_{ij}))$ contains consecutive integers. From this, together with statements (5) and (6), we obtain $f = f_{zu} = f_{xy}$. □

COROLLARY 2.5. *For any tree T there is an optimal labeling f_{xy} in which both x and y are leaves.*

From now on, all optimal labelings we consider are as specified in Corollary 2.5. The path $\alpha(x, y)$ is called a *basic path* for $P(T)$.

Theorem 2.4 and (2.1) tell us that in order to find the profile of a tree T we need only find a basic path $\alpha(x, y)$ whose deletion results in a forest with the smallest possible profile.

For technical reasons, we now consider the following restricted path deletion problem. Suppose y is a fixed vertex in tree T; find a path $\alpha(x, y)$ ending at y such that $P(T - \alpha(x, y))$ is minimum. We use $P'(T, y)$ to denote this minimum value. We also call $\alpha(x, y)$ the *basic path* for $P'(T, y)$.

Suppose f_{xy} is an optimal labeling of T and the tree T is as shown in Fig. 2.1. Denote by ${}^kT$ (respectively T^k) the subtree of T that contains $v_0, v_1, \ldots, v_k, F_1, \ldots, F_k$ (respectively $v_k, \ldots, v_r, F_k, \ldots, F_r$). From Theorem 2.4 and (2.1), we obtain the following corollary.

COROLLARY 2.6. *For a basic path $(v_0, v_1, \ldots, v_r)$ for $P(T)$, the following hold:*

(1) $(v_0, v_1, \ldots, v_k)$ *is a basic path for* $P'({}^kT, v_k)$ *and* $P'({}^kT, v_k) = \sum_{i=1}^{k} P(F_i) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} P(T_{ij})$ *for* $1 \le k \le r$.

(2) $(v_k, v_{k+1}, \ldots, v_r)$ *is a basic path for* $P'(T^k, v_k)$ *and* $P'(T^k, v_k) = \sum_{i=k}^{r} P(F_i) = \sum_{i=k}^{r} \sum_{j=1}^{n_i} P(T_{ij})$ *for* $1 \le k \le r$.

(3) $P(T) = |E(T)| + P'({}^sT, v_s) + P'(T^t, v_t) + \sum_{i=s+1}^{t-1} P(F_i)$ *for* $1 \le s < t \le r$.

PROPOSITION 2.7. $P(T) \le P'(T, y) + |E(T)|$ *for any vertex y in T.*

3. Main theorems. This section develops theorems that restrict the possibilities of the basic paths for $P(T)$ and $P'(T, y)$. In particular, the basic path $\alpha(x, y)$ for $P(T)$ contains the centroids of T. We also prove that the basic path for $P'(T, u)$ is either $\alpha(x, u)$ or $\alpha(y, u)$, and the deletion of the basic path for $P'(T, u)$ from T results in a forest each of whose components has at most $2|V(T)|/3$ vertices. These results are the keystone of our algorithm for the profile maximization problem.

A *centroid* of a tree of n vertices is a vertex whose deletion results in a forest each of whose components has at most $\lfloor \frac{n}{2} \rfloor$ vertices. It is well known that a tree has either exactly one centroid or exactly two adjacent centroids (see [2]). A "from leaves to center" method can be employed to derive the centroids of a tree. This method requires linear time.

THEOREM 3.1. *Any basic path $\alpha(x, y)$ for $P(T)$ contains all centroids of T.*

Proof. Suppose there is a centroid of T not in the basic path

$$\alpha(x, y) = (x, \ldots, v_1, u, v_2, \ldots, y).$$

Then T is of the form shown in Fig. 3.1, with $|V(T')| \ge n/2$ where $n = |V(T)|$. By Corollary 2.6 (3), we have

$$(3.1) \qquad P(T) = |E(T)| + P'(T_1, v_1) + P'(T_2, v_2) + \sum_{i=3}^{k} P(T_i) + P(T').$$

Up to a symmetric argument, we may assume that $|V(T_1)| \le |V(T_2)|$. Let $\alpha(z, v)$ be a basic path for $P(T')$. Corollary 2.6 (3) and Proposition 2.2 give

$$(3.2) \qquad P(T') \ge |E(T')| + P'(T_a, a) + P'(T_b, b) + \sum_{j=1}^{m} P(F_j).$$

We also assume that $|V(T_a)| \le |V(T_b)|$. Now consider the labeling f_{vy} for T. By (2.1) and Corollary 2.6, we have

(3.3)

$$P_{f_{vy}}(T) = |E(T)| + P'(T_b, b) + P'(T_2, v_2) + P(T_a) + \sum_{j=1}^{m} P(F_j) + P(T_1) + \sum_{i=3}^{k} P(T_i).$$

Equations (3.1) to (3.3) together lead to that $|E(T')| \le P(T_a) - P'(T_a, a) + P(T_1) - P'(T_1, v_1)$. Then $|E(T')| \le |E(T_a)| + |E(T_1)|$ by Proposition 2.7. Thus

$$\begin{aligned} |E(T_1)| &\ge |E(T')| - |E(T_a)| \\ &> |E(T')|/2 \qquad (\text{since } |E(T_a)| \le |E(T_b)| \text{ and } T' - (T_a \cup T_b) \ne \emptyset) \\ &\ge (n - |E(T')|)/2 \qquad (\text{since } |E(T')| \ge n/2) \\ &\ge |E(T_1)| \qquad (\text{since } |E(T_1)| \le |E(T_2)|), \end{aligned}$$

which is a contradiction. □

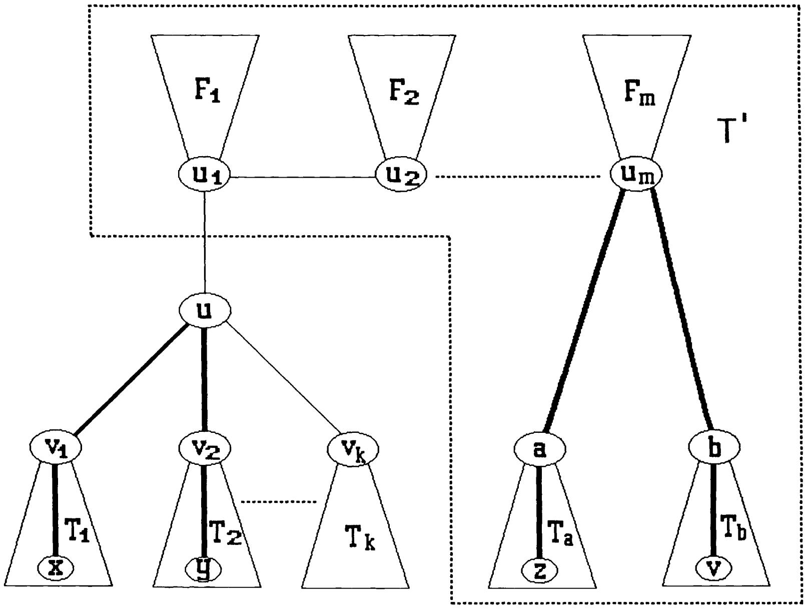


FIG. 3.1.

Similar arguments lead to the following theorem.

THEOREM 3.2. *Suppose y is a fixed vertex of a tree T of n vertices. For any basic path $\alpha(xy)$ of $P'(T, y)$, every component of $T - \alpha(x, y)$ has at most $2n/3$ vertices.*

Proof. The proof of this theorem is exactly the same as that for Theorem 3.1, except now we assume $|E(T')| > 2n/3$ and $|E(T_a)| \le |E(T_b)|$, and there is no assumption that $|E(T_1)| \le |E(T_2)|$. However, we still have $|E(T')| \le |E(T_a)| + |E(T_1)|$. Then

$$\begin{aligned} |E(T_1)| &\ge |E(T')| - |E(T_a)| \\ &\ge |E(T')|/2 \qquad (\text{since } |E(T_a)| \le |E(T_b)|) \\ &> n - |E(T')| \qquad (\text{since } |E(T')| > 2n/3) \\ &\ge |E(T_1)|, \end{aligned}$$

which is a contradiction. □

Figure 3.2 gives an example in which a basic path $\alpha(x, y)$ for $P'(T, y)$ does not contain the centroid z of T.

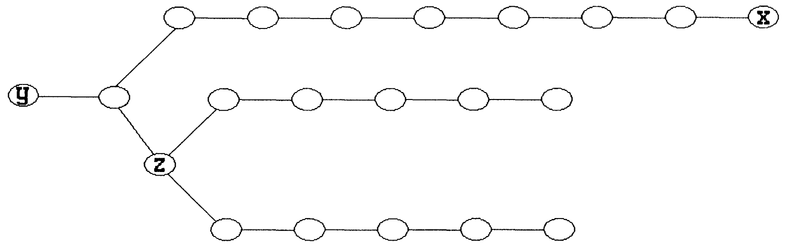


FIG. 3.2.

THEOREM 3.3. *If $\alpha(x, y)$ is a basic path for $P(T)$ and u is a fixed vertex in T, then either $\alpha(x, u)$ or $\alpha(y, u)$ is a basic path for $P'(T, u)$.*

Proof. Suppose $\alpha(x, y) = (x, \ldots, v_1, u_1, v_2, \ldots, y)$ and $(u_1, u_2, \ldots, u_r = u)$ is the unique path from $\alpha(x, y)$ to u, as shown in Fig. 3.3. Let $\alpha(z, u)$ be a basic path for $P'(T, u)$.

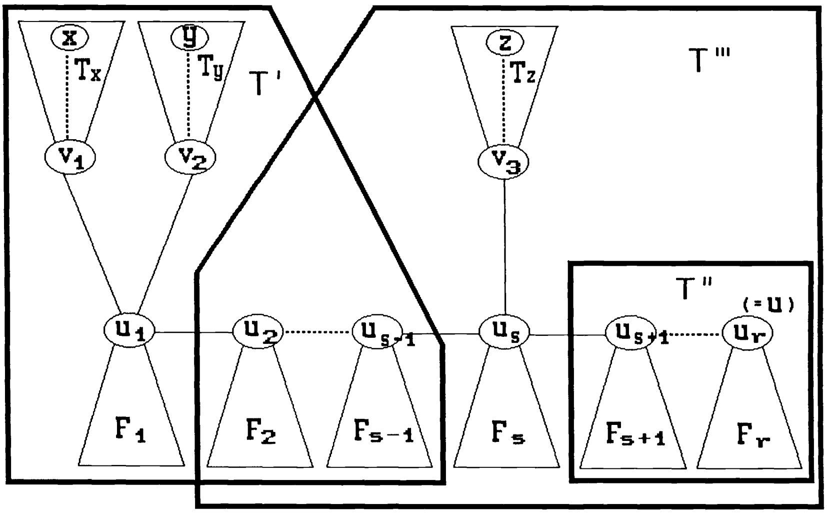


FIG. 3.3.

Case 1. $z \in V(T_x)$. In this case, $u_s = u_1$, $v_3 = v_1$, and $T_z = T_x$. By Corollary 2.6 (1), $\alpha(x, v_1)$ is a basic path for $P'(T_x, v_1)$, and so $P(T_x - \alpha(x, v_1)) \leq P(T_z - \alpha(z, v_3))$. Then

$$\begin{aligned}P'(T, u) &= P(T - \alpha(z, u))\\ &= P(T_z - \alpha(z, v_3)) + P(T_y) + \sum_{i=1}^{r} P(F_i)\\ &\geq P(T_x - \alpha(x, v_1)) + P(T_y) + \sum_{i=1}^{r} P(F_i)\\ &= P(T - \alpha(x, u)).\end{aligned}$$

Hence $\alpha(x, u)$ is also a basic path for $P'(T, u)$.

Case 2. $z \in V(T_y)$. By a similar argument, $\alpha(y, u)$ is also a basic path for $P'(T, u)$.

Case 3. $z \notin V(T_x)$ and $z \notin T(T_y)$. Let T', T'', and T''' be subtrees, as shown in Fig. 3.3. Note that in the case of $s = 1$, $T' = T_x \cup T_y$ is not a tree. Now

$$P(T - \alpha(z, u)) = P'(T_z, v_3) + P(T') + \sum_{i=s}^{r} P(F_i). \tag{3.4}$$

Note that $P'(T_z, v_3) = P(T_z - \alpha(z, v_3))$. By Proposition 2.2, we have

$$P(T') \geq P(T_x) + P(T_y) + \sum_{i=1}^{s-1} P(F_i). \tag{3.5}$$

Since $\alpha(x, y)$ is a basic path for $P(T)$, we have $P_{f_{xz}}(T) \geq P_{f_{xy}}(T)$. By (2.1) and Corollary 2.6 (3) we have

$$\begin{aligned} &|E(T)| + P(T_x - \alpha(x, v_1)) + P(T_y) + \sum_{i=1}^{s} P(F_i) + P(T'') + P(T_z - \alpha(z, v_3)) \\ &\geq |E(T)| + P'(T_x, v_1) + P'(T_y, v_2) + P(F_1) + P(T'''). \end{aligned} \tag{3.6}$$

Note that $P'(T_x, v_1) = P(T_x - \alpha(x, v_1))$. Again, by Proposition 2.2,

$$P(T''') \geq \sum_{i=2}^{s} P(F_i) + P(T_z) + P(T''). \tag{3.7}$$

Equations (3.4) to (3.7) together lead to

$$P(T - \alpha(z, u)) \geq P'(T_y, v_2) + P(T_x) + \sum_{i=1}^{r} P(F_i) + P(T_z) = P(T - \alpha(y, u)).$$

Hence $\alpha(y, u)$ is a basic path for $P'(T, u)$. □

4. The algorithm. We can use the theorems in §3 to design an efficient algorithm for the profile minimization problem in a tree T. By Theorem 3.1, the basic idea of our algorithm is to find a centroid z first in linear time. Suppose $T - z = \cup_{1 \leq i \leq m} T_i$, where u_i is the only vertex of T_i that is adjacent to z in T (see Fig. 4.1). To use Corollary 2.6 (3), we need to find all

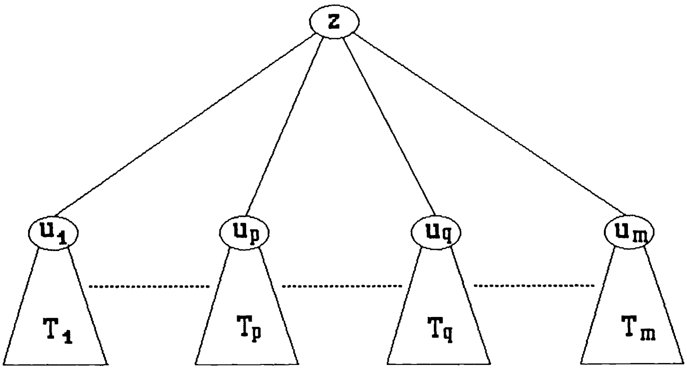


FIG. 4.1.

profiles $P(T_i)$ and $P'(T_i, u_i)$ recursively. In the following, Algorithm PROFILE finds $P(T)$ and Algorithm PROFILE1 finds $P'(T, u)$. Note that, in order to make use of Theorem 3.3, Algorithm PROFILE not only has to output the value $P(T)$ but also a basic path.

ALGORITHM PROFILE
Input: A tree T of n vertices.
Output: A basic path $\alpha(x, y) = (v_0, v_1, \ldots, v_r)$ for $P(T)$ and the values $P(T)$ and $P(T_{ij})$ for $1 \le i \le r-1$ and $1 \le j \le n_i$.
Method:

```
1. find a centroid $z$ of $T$.
2. let $T - z = \cup_{1 \le k \le m} T_k$ and $z$ be adjacent to $u_k \in V(T_k)$ for $1 \le k \le m$.
3. for each $1 \le k \le m$, recursively call PROFILE for $T_k$ to get a basic path $\alpha(x_k, y_k)$
   and values $P(T_k)$ and $P(T_{kij})$, where $T_{kij}$ are the components of $T_k - \alpha(x_k, y_k)$.
4. for each $1 \le k \le m$, recursively call PROFILE1 for $(T_k, u_k)$ to get a basic path
   $\alpha(z_k, u_k)$ and values $P'(T_k, u_k)$ and $P(T'_{kij})$, where $T'_{kij}$ are the components of $T_k -$
   $\alpha(z_k, u_k)$.
5. let $P(T) = n + \min_{1 \le p < q \le m}\{P'(T_p, u_p) + P'(T_q, u_q) + \sum_{i \ne p,q}(T_i)\}$, where $p^*$ and
   $q^*$ attain the above minimum.
6. let $\alpha(x, y) = \alpha(z_{p^*}, u_{p^*}) + z + \alpha(u_{q^*}, z_{q^*})$.
7. combine profiles $P(T_{p^*ij})$, $P(T_k)$ for $k \ne p^*, q^*$, and $P(T_{q^*ij})$ to get profiles $P(T_{ij})$.
```

To find $P'(T, u)$, we note that by Theorem 3.3, either $\alpha(x, u)$ or $\alpha(y, u)$ is a basic path for $P'(T, u)$. So we consider the configuration in Fig. 3.3 with T_z omitted.

ALGORITHM PROFILE1
Input: Tree T of n vertices with a basic path $\alpha(x, y) = (v_0, v_1, \ldots, v_r)$ for $P(T)$ and the values $P(T_{ij})$ for $1 \le i \le r-1$ and $1 \le j \le n_i$. u is a fixed vertex in T.
Output: A basic path $\alpha(z, u) = (v'_0, v'_1, \ldots, v'_{r'})$ for $P'(T, u)$ and the values $P'(T, u)$ and $P(T'_{ij})$ for $1 \le i \le r'$ and $1 \le j \le n'_i$.
Method:

```
1. identify the path $(u_1, u_2, \ldots, u_r)$ as in Fig. 3.3.
2. recursively use PROFILE to solve $P(T_x)$, $P(T_y)$, $P(F_i)$ (in fact $P(T_{ij})$ for each
   component in $F_i$) for $1 \le i \le r$.
3. $a = P'(T_x, v_1) + P(T_y) + \sum_{i=1}^{r} P(F_i)$,
   $b = P'(T_y, v_2) + P(T_x) + \sum_{i=1}^{r} P(F_i)$,
   where $P'(T_x, v_1)$ and $P'(T_y, v_2)$ can be computed from the input values $P(T_{ij})$.
4. $P'(T, u) = \min\{a, b\}$.
   if $a \le b$ then $z = x$ else $z = y$.
5. combine part of the profiles $P(T_{ij})$, $P(T_x)$, or $P(T_y)$, and $P(F_i)$ to get profiles $P(T'_{ij})$.
```

5. Time complexity. This section shows that the time complexities of the above two algorithms are $O(n^{1.722})$. Let $f(n)$ (respectively, $g(n)$) be the time complexity for Algorithm PROFILE (respectively, PROFILE1).

In Algorithm PROFILE, Step 3 (respectively, 4) needs $\sum_{i=1}^{m} f(n_i)$ (respectively, $\sum_{i=1}^{m} g(n_i)$) time, where $n_i = |V(T_i)|$ for $1 \le i \le m$. All other steps need $O(n)$ time. Note that for Step 5 we only have to find the smallest and the second smallest values of $P'(T_i, w_i) - P(T_i)$. Therefore

$$(5.1) \qquad f(n) = \sum_{i=1}^{m}\{f(n_i) + g(n_i)\} + c_1 n,$$

where

$$\sum_{i=1}^{m} n_i = n - 1 \quad \text{and} \quad n_i \le n/2 \quad \text{for} \quad 1 \le i \le m.$$

Similarly, in Algorithm PROFILE1, Step 2 needs $\sum_{i=1}^{m} f(n_i)$ time and all other steps need $O(n)$ time. Thus, by Theorem 3.2,

$$g(n) = \sum_{i=1}^{m} f(n_i) + c_2 n, \tag{5.2}$$

where

$$\sum_{i=1}^{m} n_i \le n - 1 \quad \text{and} \quad n_i \le 2n/3 \quad \text{for} \quad 1 \le i \le m.$$

To solve (5.1) and (5.2), we first choose a number $\sigma > 1$, which is very close to 1, say, $\sigma = 1.001$. Then choose $1 < \lambda < 2$ such that

$$\epsilon = \left(\frac{2}{3}\right)^{\lambda} + \left(\frac{1}{3}\right)^{\lambda} < 1$$

and

$$\delta = (1 + \sigma\epsilon)2\left(\frac{1}{2}\right)^{\lambda} < 1.$$

Note that a simple computer program gives that $\lambda = 1.722$ for $\sigma = 1.001$.

THEOREM 5.1. *There exists a constant c such that $f(n) \le cn^{\lambda}$ and $g(n) \le c\sigma\epsilon n^{\lambda}$ for all n, i.e., $f(n) = O(n^{\lambda})$ and $g(n) = O(n^{\lambda})$.*

Proof. The proof is by induction on n. Assume that there exists a constant c such that $f(n') \le cn'^{\lambda}$ and $g(n') \le c\sigma\epsilon n'^{\lambda}$ for all $n' < n$. We also assume that $c \ge c_1/(1-\delta)$ and $c \ge c_2/(\sigma - 1)\epsilon$.

For $0 < a \le b$, consider the function $h(x) = (b+x)^{\lambda} + (a-x)^{\lambda} - b^{\lambda} - a^{\lambda}$ where $0 \le x \le a$. Note that $h'(x) = \lambda(b+x)^{\lambda-1} - \lambda(a-x)^{\lambda-1} \ge 0$. So h is an increasing function and then $h(x) \ge h(0) = 0$ for $0 \le x \le a$. Thus

$$b^{\lambda} + a^{\lambda} \le (b+x)^{\lambda} + (a-x)^{\lambda} \quad \text{for} \quad 0 \le x \le a \le b. \tag{5.3}$$

By (5.1) and the induction hypothesis, we have $f(n) \le c(1+\sigma\epsilon)\sum_{i=1}^{m} n_i^{\lambda} + c_1 n$ where $\sum_{i=1}^{m} n_i = n - 1$ and $n_i \le n/2$ for $1 \le i \le m$. Repeatedly apply (5.3) to get $\sum_{i=1}^{m} n_i^{\lambda} \le \left(\frac{n}{2}\right)^{\lambda} + \left(\frac{n}{2} - 1\right)^{\lambda}$. Therefore $f(n) \le c(1+\sigma\epsilon)2\left(\frac{1}{2}\right)^{\lambda} n^{\lambda} + c_1 n = c\delta n^{\lambda} + c_1 n$. By the choice of c, $c_1 n \le c(1-\delta)n \le c(1-\delta)n^{\lambda}$. Then $f(n) \le cn^{\lambda}$.

By (5.2) and the induction hypothesis, we have $g(n) \le c\sum_{i=1}^{m} n_i^{\lambda} c_2 n$ where $\sum_{i=1}^{m} n_i \le n - 1$ and $n_i \le 2n/3$ for $1 \le i \le m$. Repeatedly apply (5.3) to get $\sum_{i=1}^{m} n_i^{\lambda} \le (2n/3)^{\lambda} + \left(\frac{n}{3}\right)^{\lambda} = \epsilon n^{\lambda}$. Also, by the choice of c, $c_2 n \le c(\sigma - 1)\epsilon n \le c(\sigma - 1)\epsilon n^{\lambda}$. Then $g(n) \le c\epsilon n^{\lambda} + c(\sigma - 1)\epsilon n^{\lambda} = c\sigma\epsilon n^{\lambda}$. □

Acknowledgments. We thank the two anonymous referees for many useful suggestions on the revision of this paper.

REFERENCES

[1] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403–423.

[2] F. BUCKLEY AND F. HARARY, *Distance in Graphs*, Addison-Wesley, Reading, MA, 1990.

[3] F. R. K. CHUNG, *On optimal linear arrangements of trees*, Comput. Math. Appl., 10 (1984), pp. 43–60.

[4] M. R. GAREY AND D. S. JOHNSON, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.

[5] Y. LIN AND J. YUAN, *Profile minimization problem for matrices and graphs*, preprint, Dept. of Math., Zhengzhou University, People's Republic of China, 1990.

[6] ———, *Minimum profile of grid networks in structure analysis*, to appear.

[7] Y. SHILOACH, *A minimum linear arrangement algorithm for undirected tree*, SIAM J. Comp., 8 (1979), pp. 15–32.

[8] R. P. TEWARSON, *Sparse Matrices*, Academic Press, New York, 1973.

[9] O. C. ZIENKIEWICZ, *Finite Element Method in Engineering Science*, McGraw Hill, London, 1971.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 82–96, February 1994

007

MODIFICATIONS OF COMPETITIVE GROUP TESTING*

D.-Z. DU[†‡], G.-L. XUE[†], S.-Z. SUN[†] AND S.-W. CHENG[†]

Abstract. Many fault-detection problems fall into the following model: There is a set of n items, some of which are defective. The goal is to identify the defective items by using the minimum number of tests. Each test is on a subset of items and tells whether the subset contains a defective item or not. Let $M_\alpha(d, n)(M_\alpha(d \mid n))$ denote the maximum number of tests for an algorithm α to identify d defectives from a set of n items provided that d, the number of defective items, is known (unknown) before the testing. Let $M(d, n) = \min_\alpha M_\alpha(d, n)$. An algorithm α is called a *competitive algorithm* if there exist constants c and a such that for all $n > d > 0$, $M_\alpha(d \mid n) \leq cM(d, n) + a$. This paper confirms a recent conjecture that there exists a bisecting algorithm A such that $M_A(d \mid n) \leq 2M(d, n) + 1$. Also, an algorithm B such that $M_B(d \mid n) \leq 1.65M(d, n) + 10$ is presented.

Key words. group testing, competitive algorithm

AMS subject classification. 68P10

1. Introduction.

A feature of on-line problems [12], [15] is that information about input is not completely given at the beginning but is collected during the process of seeking a solution. This feature makes an optimal solution very hard to attain; an option is to consider competitive algorithms, which take responsibility for producing a reasonable solution. A similar situation occurs in some searching problems.

Consider a set of n items. Some items are *defective*, and others are *good*. The problem is to identify the defective items by a sequence of tests. Each test is on a subset of items and tells us whether the subset contains a defective item or not. In the former case the subset is said to be *contaminated*, and in the latter case the subset is said to be *pure*. The problem has applications in high-speed computer networks [4], string pattern recognition [11], medical examination [5], and quantity searching [3]. It also occurs in statistics [16], information theory [8], and related areas [1], [2], [10]. In the literature, the problem has been named *group testing*. It has two categories based on whether the tests have errors or are error free [3], [14]. In this paper we study error-free tests.

A classic model for such a searching problem is to assume that the number of defective items is known. This assumption is somewhat artificial since in practice this number is usually unknown *a priori* and it can be known only after testing. If the number of defective items is unknown at the beginning, how do we design and analyze algorithms? Motivated by the study of on-line problems [12], [15], Du and Hwang [7] proposed the concept of competitive algorithms for the group-testing problem.

Let $N_\alpha(s \mid d, n)(N_\alpha(s \mid n))$ be the number of tests that an algorithm α spends on a sample s of n items under the condition that d, the number of defective items, is known (unknown). Denote

$$M_\alpha(d, n) = \max_{s \in \mathcal{A}(n,d)} N_\alpha(s \mid d, n),$$

$$M(d, n) = \min_\alpha M_\alpha(d, n),$$

$$M_\alpha(d \mid n) = \max_{s \in \mathcal{A}(n,d)} N_\alpha(s \mid n),$$

*Received by the editors March 16, 1991; accepted for publication (in revised form) August 4, 1992. Supported in part by NSF grant CCR-9208913.

[†] Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

[‡] Also at Institute of Applied Mathematics, Chinese Academy of Sciences, Beijing, China.

when $\mathcal{A}(n, d)$ is the set of samples of n items containing d defective items. An algorithm α is called a *c-competitive* algorithm if there exists a constant a such that for $0 < d < n$, $M_\alpha(d \mid n) \leq c \cdot M(d, n) + a$. Note that in the definition we exclude two cases, $d = 0$ and $d = n$, because $M(0, n) = M(n, n) = 0$. A c-competitive algorithm for a constant c is simply called a competitive algorithm, and c is called the *competitive ratio* of the algorithm.

Du and Hwang [7] proposed a bisecting algorithm with competitive ratio 2.75 and conjectured that there exists a bisecting algorithm A such that $M_A(d \mid n) \leq 2M(d, n) + 1$ for $1 \leq d \leq n - 1$. Soon thereafter, Bar-Noy, Hwang, Kessler, and Kutten [4] discovered a doubling algorithm D such that $M_D(d \mid n) \leq 2M(d, n) + 5$ for $1 \leq d \leq n - 1$. In this paper we confirm Du and Hwang's conjecture by presenting a bisecting algorithm A such that $M_A(d \mid n) \leq 2M(d, n) + 1$. Also, we present an algorithm B such that $M_B(d \mid n) \leq 1.65M(d, n) + 10$.

2. Preliminaries. The analysis of competitive group testing involves both lower-bound and upper-bound problems. In this section we list some results about the lower bound for $M(d, n)$ that will be used in other sections.

There are $\binom{n}{d}$ samples for n items containing d defectives. Each test divides those samples into two sets. Therefore, we have the following information lower bound.

LEMMA 2.1. *For* $n > d > 0$

$$M(d, n) \geq \left\lceil \log_2 \binom{n}{d} \right\rceil \geq d \log_2 \frac{n}{d}.$$

A more useful bound derived from the aforementioned information lower bound is stated in the next lemma.

LEMMA 2.2. *For* $0 < d < \rho n$, $\rho \in (0, 1)$,

$$M(d, n) \geq d \left(\log_2 \frac{n}{d} + \log_2(e\sqrt{1 - \rho}) \right) - 0.5 \log_2 d - 0.5 \log_2(1 - \rho) - 1.568.$$

Proof. Note that the information lower bound is $\log_2 \binom{n}{d}$ for $M(d, n)$. since $n/d < (n - i)/(d - i)$ for $0 < i < d$, we have $M(d, n) \geq d \log_2(n/d)$. Now, we use Stirling's formula, $n! = \sqrt{2\pi n}(n/e)^n e^{\epsilon/(12n)} (0 < \epsilon < 1)$ [13], to obtain the following estimation.

$$\begin{aligned}\binom{n}{d} &\geq \sqrt{\frac{n}{2\pi d(n - d)}} \left(\frac{n}{d}\right)^d \left(\frac{n}{n - d}\right) n - d \exp\left(-\frac{1}{12d} - \frac{1}{12(n - d)}\right) \\ &> \frac{1}{\sqrt{d}} \left(\frac{n}{d}\right)^d \left[\left(1 + \frac{d}{n - d}\right)^{(n-d)/d+0.5}\right]^d \left(\frac{n}{n - d}\right)^{-0.5d+0.5} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{6}\right) \\ &> \frac{1}{\sqrt{d}} \left(\frac{n}{d}\right)^d (e\sqrt{1 - \rho})^d (1 - \rho)^{-0.5} 2^{-1.568}.\end{aligned}$$

Thus

$$M(d, n) > d \left(\log_2 \frac{n}{d} + \log_2(e\sqrt{1 - \rho}) \right) - 0.5 \log_2 d - 0.5 \log_2(1 - \rho) - 1.568. \qquad \square$$

Applying Lemma 2.2 to the case of $\rho = 8.21$, we have the following.

COROLLARY 2.3. *For* $0 < d < (8/21)n$

$$M(d, n) \geq d \left(\log_2 \frac{n}{d} + 1.096 \right) - 0.5 \log_2 d - 1.222.$$

Sometimes the information lower bound is too rough. Then the bound in the following lemma may be useful. This lemma can be found in [9].

LEMMA 2.4. *For* $1 \leq d \leq n-1$

$$M(d,n) \geq \min\left\{n-1, \max_{0\leq k<d}\left[\left\lceil \log \binom{n-k}{d-k}\right\rceil + 2k\right]\right\}.$$

Hu, Hwang, and Wang [9] conjectured that for $n \leq 3d$, $M(d,n) = n-1$. Du and Hwang [6] obtained the following from Lemma 2.4.

LEMMA 2.5. *If* $n > d \geq (8/21)n$, *then* $M(d,n) = n-1$.

3. Tight bound for bisecting. Let S be a set of n items. The principle of the bisecting algorithm is that at each step, if a contaminated subset X of S is discovered, then X is bisected and the resulting two subsets X' and X'' are tested. The method of bisecting will affect the competitive ratio. Here, we choose X' to contain $2^{\lceil \log_2 |X| \rceil - 1}$ items and $X'' = X \backslash X'$, where $|X|$ denotes the number of elements in X.

```
Algorithm A;
      input $S$;
      $G := \theta$; {a container of good items}
      $D := \theta$; {a container of defective items}
      TEST($S$);
      if $S$ is pure then $G := S$ and $Q := \theta$
         else $Q := \{S\}$;
      repeat
         choose the frontier element $X$ of queue $Q$;
         bisect $X$ into $X'$ and $x''$;
         TEST($X'$);
         if $X'$ is contaminated then TEST($X''$);
         {If $X'$ is pure, then it is known that $X''$ is contaminated.}
         for $Y := X'$ and $X''$ do begin
            if $Y$ is pure then $G := G \cup Y$;
            if $Y$ is a contaminated singleton then $D := D \cup Y$;
            if $Y$ is contaminated but not a singleton
               then put $Y$ into the rear of queue $Q$
         end-for;
       until $Q = \theta$
end-algorithm.
```

We will show the following.

THEOREM 3.1. *For* $1 \leq d \leq n-1$

$$M_A(d \mid n) \leq 2M(d,n) + 1. \tag{1}$$

A binary tree is a rooted tree with the property that each *internal node* has exactly two sons. A node is said to be the kth level of the tree if the path from the root to the node has length $k-1$. So the root is on the first level. Let i be the number of nodes in a binary tree, and let j be the number of internal nodes in the tree. It is well known that $i = 2j+1$.

Let T denote a binary tree with nodes denoted by X's in Algorithm A such that two nodes X' and X'' are sons of X if and only if they are obtained by bisecting X. Note that each leaf of T must identify at least one item. So T has at most n leaves. It follows that T has at most $2n-1$ nodes. Therefore, $M_A(d \mid n) \leq 2n-1$. By Lemma 2.5 if $d/n \geq 8/21$, then

$M_A(d \mid n) \le 2M(d,n) + 1$. Before proving Theorem 3.1 for $d/n < 8/21$, let us first show some lemmas.

For convenience we assume that the value of function $d \log_2 \frac{n}{d}$ at $d = 0$ is 0 because $\lim_{d \to 0} d \log_2 \frac{n}{d} = 0$. The following lemma is an important tool for our analysis.

LEMMA 3.2. *Let $d = d' + d''$ and $n = n' + n''$, where $d' \ge 0, d'' \ge 0, n' > 0$, and $n'' > 0$. Then*

$$d' \log_2 \frac{n'}{d'} + d'' \log_2 \frac{n''}{d''} \le d \log_2 \frac{n}{d}.$$

Proof. Note that $(d^2/dx^2)(-x \log_2 x) = -\frac{1}{x \ln 2} < 0$ for $x > 0$. So $-x \log_2 x$ is a concave function. Thus

$$\begin{aligned} d' \log_2 \frac{n'}{d'} + d'' \log_2 \frac{n''}{d''} &= n \left(\frac{n'}{n} \frac{d'}{n'} \log_2 \frac{n'}{d'} + \frac{n''}{n} \frac{d''}{n''} \log_2 \frac{n''}{d''} \right) \\ &\le n \left(\frac{d}{n} \log_2 \frac{n}{d} \right) \\ &= d \log_2 \frac{n}{d}. \qquad \square \end{aligned}$$

Clearly, when n is a power of 2, the analysis is relatively easy. So we first study this case.

LEMMA 3.3. *Let n be a power of 2. Then for $1 \le d \le n$*

$$M_A(d \mid n) \le 2d \left(\log_2 \frac{n}{d} + 1 \right) - 1.$$

Proof. Consider the binary tree T defined before Lemma 3.2. Clearly, every internal node must be contaminated and there exist exactly d contaminated leaves. Next, we count how many contaminated nodes we can have. Denote $u = \lceil \log_2 n \rceil$, $v = \lceil \log_2 d \rceil$, and $v' = v - \log_2 d$. Then the tree T has $u + 1$ levels, and the ith level contains 2^{i-1} nodes. Note that each level has at most d contaminated nodes and the $(v+1)st$ level is the first level that has at least d nodes. Thus the number of contaminated nodes is at most

$$\begin{aligned} \sum_{i=1}^{v} 2^{i-1} + (u - v + 1)d &= 2^v - 1 + d \left(\log_2 \frac{n}{d} + 1 - v' \right) \\ &= -1 + d \left(\log_2 \frac{n}{d} + 1 - v' + 2^{v'} \right) \\ &\le -1 + d \left(\log_2 \frac{n}{d} + 2 \right). \end{aligned}$$

The last inequality sign holds since $f(v') = -v' + 2^{v'}$ is a convex function of v' and v' is between 0 and 1. Thus T has at most $-1 + d(\log_2 \frac{n}{d} + 1)$ internal nodes and hence at most $2d(\log_2 \frac{n}{d} + 1) - 1$ nodes. $\square$

According to the way the bisection was done, each level of T contains at most one node that is a set whose size is not a power of 2. This property plays an important role in the following.

LEMMA 3.4. *For $0 \le d \le n$*

$$M_A(d \mid n) \le 2d \left(\log_2 \frac{n}{d} + 1 \right) + 1.$$

Proof. We prove the lemma by induction on n. For $n = 1$ the proof is trivial. For $n > 1$ let S be the set of n items. If $d = 0$, then one test is sufficient so that the lemma holds obviously. If $d > 0$, then we consider two cases corresponding to S' and S'' that are obtained by bisecting S.

Case 1. S' is contaminated. Since the number of items in S' is a power of 2, Algorithm A spends at most $2d'(\log_2(|S'|/d') + 1) - 1$ tests on S', where d' is the number of defective items in S'. Let d'' be the number of defective items in S''. By the induction hypothesis, Algorithm A spends at most $2d''(\log_2(|S''|/d'') + 1) + 1$ tests. Adding one for testing S, we obtain that the total number of tests is at most

$$2d'\left(\log_2\frac{|S'|}{d'} + 1\right) + 2d''\left(\log_2\frac{|S''|}{d''} + 1\right) + 1 \leq 2d\left(\log_2\frac{n}{d} + 1\right) + 1.$$

Case 2. S' is pure. In this case Algorithm A spends a test on S' and at most $2d(\log_2\frac{|S''|}{d} + 1) + 1$ tests on S''. So, adding one more test for S, we obtain that the total number of tests is at most

$$2 + 2d\left(\log_2\frac{|S''|}{d} + 1\right) + 1 = 2d\left(\log_2\frac{2|S''|}{d} + 1\right) + 1$$
$$\leq 2d\left(\log_2\frac{n}{d} + 1\right) + 1. \qquad \square$$

To prove Theorem 3.1, by the remark that we made after the proof of Theorem 3.1, we need only to consider the case for which $d/n < 8/21$. In this case, by Corollary 2.3 we have

$$M(d,n) > d\left(\log_2\frac{n}{d} + 1.096\right) - 0.5\log_2 d - 1.222.$$

Thus by Lemma 3.4

$$M_A(d\,|\,n) \leq 2M(d,n) + 2(0.5\log_2 d + 1.222 - 0.096d) + 1.$$

Let us look at the function $h(d) = 0.5\log_2 d - 0.096d$. $h'(d) = \frac{0.5}{d\cdot\ln 2} - 0.096$. So $h(d)$ is decreasing for $d \geq 8$. We want $h(d) \leq -1.222$. This yields $d \geq 41$. Therefore, for $d \geq 41$, $M_A(d\,|\,n) \leq 2M(d,n) + 1$.

Next, we use a more accurate analysis to deal with the case of $1 \leq d \leq 41$.

Define

$$f(n,d) = \frac{\binom{n}{d}\sqrt{2}}{\left(\frac{n}{d}\right)^d 2^d}. \tag{2}$$

If $f(n,d) > 1$, then from Lemma 3.4 and the information lower bound for $M(d,n)$ it is easy to see that

$$M_A(d\,|\,n) < 2M(d,n) + 2.$$

Since both sides of the inequality are integers, we have

$$M_A(d\,|\,n) \leq 2M(d,n) + 1.$$

Next, we study the case of $f(n,d) \leq 1$.

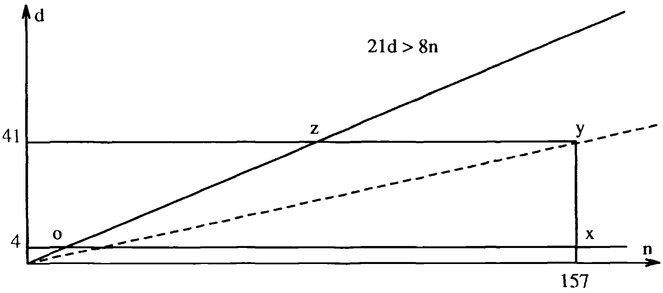


FIG. 1. *All (n, d) for $f(n, d) \leq 1$ are contained in the polygon oxyz.*

Consider the following ratio:

$$\frac{f(n, d+1)}{f(n, d)} = \frac{n-d}{2n}\left(1 + \frac{1}{d}\right)^d.$$

It is easy to see that

$$\frac{n-d}{2n} \cdot e > \frac{f(n, d+1)}{f(n, d)} > \frac{n-d}{2n}\left(\frac{d}{d+1}\right)^{1/2} \cdot e.$$

Thus we have the following lemma.

LEMMA 3.5. *For $\frac{d}{n} \geq 1 - \frac{2}{e}$, $f(n, d)$ is decreasing with respect to d. For $\frac{d}{n} \leq 1 - \frac{2}{e}\sqrt{(d+1)/d}$, $f(n, d)$ is increasing with respect to d.*

This lemma tells us the behavior of function $f(n, d)$ with respect to d. Next, we study its behavior with respect to n. Consider

$$g(n, d) = \frac{f(n+1, d)}{f(n, d)} = \frac{\frac{n+1}{n-d+1}}{\left(\frac{n+1}{n}\right)^d}.$$

Note that

$$\frac{g(n, d+1)}{g(n, d)} = \frac{n(n-d+1)}{(n+1)(n-d)} \geq 1$$

because

$$n(n-d+1) - (n+1)(n-d) = d.$$

Moreover, $g(n, 1) = 1$. Thus for $d \geq 1$, $g(n, d) \geq 1$. Therefore, we have the following lemma.

LEMMA 3.6. *For $d \geq 1$, $f(n, d)$ is increasing in n.*

From Lemmas 3.5 and 3.6 we see that if $f(n^*, d^*) > 1$, then for every $n \geq n^*$ and $(1 - \frac{2}{e}\sqrt{d/(d-1)})n + 1 \geq d \geq d^*$, $f(n, d) > 1$. Note that $f(157, 5) > 1$ and $(1 - \frac{2}{e}\sqrt{41/40}) \cdot 157 + 1 > 41$. It follows that for $n \geq 157$ and $d \geq 5$, $f(n, d) > 1$ (see Fig. 1). Unfortunately, the preceding argument does not help in the case of $d \leq 4$. In fact, it is easy to prove that for $1 \leq d \leq 4$, $f(n, d) < 1$. Actually, we need a more accurate upper bound for $M_A(d \mid n)$ for $1 \leq d \leq 4$. The details can be found in Appendix 1. Now, the remainder is a finite set of pairs (n, d)'s located in the polygon $oxyz$ as shown in Fig. 1.

For such finitely many pairs we compute $M_A(d \mid n)$ by the following formula:

$$M_A(d \mid n) = \max_{0 \le d' \le d} \{1 + M_A(d' \mid n') + M_a(d'' \mid n'')\}, \tag{3}$$

$$M_A(0 \mid n) = 1, \tag{4}$$

$$M_A(1 \mid n) = 2\lceil \log n \rceil + 1, \tag{5}$$

where $n' = 2^{\lceil \log n \rceil - 1}$, $n'' = n - n'$, and $d'' = d - d'$. We also compute a lower bound of $M(d, n)$ by the following formula:

$$\ell(d, n) = \min \left\{ n - 1, \max_{0 \le k < d} \left[\left\lceil \log \binom{n-k}{d-k} \right\rceil + 2k \right] \right\}.$$

Comparing two computational results, we find that $M_A(d \mid n) \le 2\ell(d, n) + 1$. (We can save some computation by a careful analysis; see Appendix 2 for details). This completes the proof of Theorem 3.1. □

4. 1.65-Competitive algorithm. Bar-Noy, Hwang, Kessler, and Kutten [4] discovered another way to design a competitive group-testing algorithm. Their basic idea is as follows. Because d, the number of defective items, is unknown, the algorithm tries to estimate the value of d. If d is small, the algorithm would like to find large pure sets, whereas if d is large, the algorithm would like to find small contaminated sets. To produce this behavior, the algorithm uses a doubling strategy. It tests a disjoint set of size $1, 2, \ldots, 2^i$ until a contaminated set is found. Namely, the first i sets are pure and the last set is contaminated. So the algorithm finds $1 + 2 + \cdots + 2^{i-1} = 2^i - 1$ good items and a contaminated set of size 2^i by using $i + 1$ tests. Next, the algorithm identifies a defective item from the contaminated item by a binary search with i tests as follows.

```
**Procedure** DIG($X$);
    **repeat**
        $X' := \lceil |Y|/2 \rceil$ items from $X$;
          TEST($X'$);
          **if** $X'$ is contaminated **then** $X := X'$
             **else** $X := X \backslash X'$;
                  $S := S \backslash X'$;
                  $G := G \cup X'$;
    **until** $X$ is a singleton;
    $S := S \backslash X$;
    $D := D \cup X$;
**end-procedure**;
```

Thus, the algorithm used a total of $2i + 1$ tests and identified 2^i items.

Here, we introduce a new idea as follows: Instead of testing disjoint sets of size $1, \ldots, 2^i$, the algorithm tests disjoint sets of size $1+2$, $4+8, \ldots, 2^i + 2^{i+1}$ for even i until a contaminated set is found. In this way the algorithm detects $2^i - 1$ good items by using $i/2$ tests instead of i tests. However, it found a contaminated set of size $3 \cdot 2^i$ instead of 2^i, which requires only one more test on a subset of size 2^i in order to reduce the contaminated set to either size 2^i or size 2^{i+1} with 2^i more good items. This idea is an extension of an idea given in [4] for combining the first two tests to further combining.

Let us first describe a procedure for three items, which is given in [4]. The input for this procedure is a contaminated set of three items. With two tests the procedure identifies either two defective items or at least one good item and one defective item.

```
Procedure 3-TEST({$x$, $y$, $z$});
    TEST($x$);
    TEST($y$);
    if $x$ is defective then $D := D \cup \{x\}$
        else $G := G \cup \{x\}$;
    if $y$ is defective then $D := D \cup \{y\}$
        else $G := G \cup \{y\}$;
    if $x$ and $y$ both are good
        then $S := S \backslash \{x, y, z\}$
             $D := D \cup \{z\}$
        else $S := S \backslash \{x, y\}$;
end-procedure;
```

An extension of Procedure 3-TEST is as follows. The input is a contaminated set of $3 \cdot 2^k$ items ($k > 0$). The procedure first finds either a contaminated set of size 2^k or a pure set of size 2^k and a contaminated set of size 2^{k+1} and then digs out a defective item from the resultant contaminated set.

```
Procedure 3-SET-TEST($X$);
    $X' := \min(2^k, |X|)$ items from $X$;
    TEST($X'$);
    if $X'$ is contaminated
        then $X := X'$
        else $X := X \backslash X'$
             $G := G \cup X'$
             $S := S \backslash X'$;
    DIG($X$);
end-procedure
```

Now, we describe the main body of the algorithm.

```
Algorithm B;
    input $S$;
    $D := \emptyset$;
    $G := \emptyset$;
    while $|S| \geq 3$ do
        $k := 0$;
        repeat {jumping process}
            $X := \min(2^k + 2^{k+1}, |S|)$ items from $S$;
            TEST($X$);
            if $X$ is pure then $G := G \cup X$
                                $S := S \backslash X$
                                $k := k + 2$;
            if $k = 10$ then TEST($S$)
                             if $S$ is pure then $G := G \cup S$
                                                 $S := \emptyset$;
            until $X$ is contaminated;
            if $k = 0$ then 3-TEST($X$);
            if $k > 0$ then 3-SET-TEST($X$);
        end-while;
        while $S \neq \emptyset$ do
            $x :=$ an item from $S$;
            TEST($x$);
            if $x$ is good then $G := G \cup \{x\}$;
            if $x$ is defective then $D := D \cup \{x\}$;
            $S := S \backslash \{x\}$;
        end-while;
end-algorithm
```

Next, we analyze Algorithm B in a way similar to that in §3.

LEMMA 4.1. $M_B(d\,|\,n) \leq 1.65d(\log_2 \frac{n}{d} + 1.031) + 5$.

Proof. We prove the lemma by induction on d. For $d = 0$, since the algorithm will test S when $k = 10$, it takes at most five tests to find that S is pure. Thus $M_B(0\,|\,n) \leq 5$.

For $d > 0$ suppose that the first time that the computation goes out from the jumping process is with $k = i$. So a contaminated set X of size at most $2^i + 2^{i+1}$ (i is even) and $2^i - 1$ good items are found with $i/2 + 1$ tests. Next, consider three cases.

Case 1. $i = 0$. Procedure 3-TEST identifies either two defective items or at least one good item and one defective item by two tests. Applying the induction hypothesis to the remaining $n - 2$ items, we obtain that in the former subcase the total number of tests is at most

$$\begin{aligned} &3 + 1.65(d-2)\left(\log_2 \frac{n-2}{d-2} + 1.031\right) + 5 \\ &= 1.5 \cdot 2\left(\log_2 \frac{2}{2} + 1\right) + 1.65(d-2)\left(\log_2 \frac{n-2}{d-2} + 1.031\right) + 5 \\ &\leq 1.65d\left(\log_2 \tfrac{n}{d} + 1.031\right) + 5; \end{aligned}$$

in the latter subcase the total number of tests is at most

$$\begin{aligned} &3 + 1.65(d-1)\left(\log_2 \frac{n-2}{d-1} + 1.031\right) + 5 \\ &= 1.5(\log_2 2 + 1) + 1.65(d-1)\left(\log_2 \frac{n-2}{d-1} + 1.031\right) + 5 \\ &\leq 1.65d\left(\log_2 \frac{n}{d} + 1.031\right) + 5. \end{aligned}$$

Case 2. $2 \leq i \leq 8$. Procedure 3-SET-TEST finds either one defective item by using at most $i + 1$ tests or one defective item and 2^i good items by using at most $i + 2$ tests. In the former subcase the total number of identified items is 2^i and the total number of tests for detecting them is at most

$$(i/2 + 1) + (i + 1) \leq 1.65(\log_2 2^i + 1.031);$$

in the latter case the total number of identified items is 2^{i+1} and the total number of tests for identifying them is at most

$$(i/2+1)+(i+2) \leq 1.50(\log_2 2^{i+1}+1).$$

Applying the induction hypothesis to the remaining unidentified items and using Lemma 3.2, we can obtain the upper bound $1.65d(\log_2 \frac{n}{d}+1.031)+5$ for the total number of tests.

Case 3. $i \geq 10$. This case is similar to Case 2. The difference is that the algorithm uses one more test for testing on S when $i = 10$. So, corresponding to the two subcases in Case 2, we have that in the former subcase the total number of tests for identifying $2^i - 1$ good items and one defective item is at most

$$(i/2+1)+(i+2) \leq 1.65(\log_2 2^i + 1.031);$$

in the latter subcase the total number of tests for identifying 2^{i+1} good items and one defective item is at most

$$(i/2+1)+(i+3) \leq 1.65(\log_2 2^{i+1}+1.031).$$

The proof is completed by applying the induction hypothesis to the remaining unidentified items and using Lemma 3.2. □

LEMMA 4.2. *For* $0 \leq d \leq n$, $M_B(d \mid n) \leq 1.5n$.

Proof. We prove the lemma by induction on d. For $d = 0$ the algorithm needs one test when $n \leq 3$, two tests when $4 \leq n \leq 15$, and at most five tests when $n > 16$, so that $M_B(0 \mid n) \leq 1.5n$. For $d > 0$ suppose that the first time that the computation goes out from the jumping process is with $k = i$. So a contaminated set X of size at most $2^i + 2^{i+1}$ (i is even) and $2^i - 1$ good items are found with $i/2+1$ tests. Next, we follow the trace of the proof of the last lemma to verify that in each case the number of tests is at most 1.5 times the number of identified items.

Case 1. $i = 0$. Two items were identified by using three tests.

Case 2. $2 \leq i \leq 8$. Either 2^i items were identified by using $1.5i+2$ tests or 2^{i+1} items were identified by using $1.5i+3$ tests. Since $i \geq 2$, $1.5i+2 \leq 1.5 \cdot 2^i$ and $1.5i+3 \leq 1.5 \cdot 2^{i+1}$.

Case 3. $i \geq 10$. Either 2^i items were identified by using $1.5i+3$ tests or 2^{i+1} items were identified by using $1.5i+4$ tests. Since $i \geq 10$, $1.5i+3 \leq 1.5 \cdot 2^i$ and $1.5i+4 \leq 1.5 \cdot 2^{i+1}$.

The proof is completed by applying the induction hypothesis to the remaining unidentified items and adding the bound to the inequality in each of the cases or subcases. □

THEOREM 4.3. *For* $1 \leq d \leq n-1$, $M_B(d \mid n) \leq 1.65M(d,n)+10$.

Proof. If $d/n \geq 8/21$, then $M(d,n) = n-1$. The theorem then follows from Lemma 4.2. If $d/n < 8/21$, then by Lemma 4.1 and Corollary 2.3 we have

$$M_B(d \mid n) \leq 1.65M(d,n)+5+1.65(0.5\log_2 d - 0.065d+1.222).$$

Denote $h(d) = 0.5\log_2 d - 0.065d$. Then $h(d)$ increases for $d \leq 11$ and decreases for $d \geq 12$. Thus $h(d) \leq \max(h(11), h(12)) \leq 1.015$. Hence $M_B(d \mid n) \leq 1.65M(d,n)+9.3$. □

By modifying Algorithm B, the competitive ratio could be further improved to approach 1.5. The modification can be done through studying the competitive group testing for a small number of items. For example, if instead of Procedure 3-TEST we use a procedure for testing 12 items, the competitive ratio can be decreased to be less than 1.6. However, how to push the competitive ratio down from 1.5 is unknown.

Appendix 1: Proof of Theorem 3.1 for $1 \le d \le 4$. Let us prove the following lemmas.

LEMMA 3.7.

$$\begin{aligned} M_A(1\,|\,2^u) &= 2u+1 \quad \textit{for } u \ge 0, \\ M_A(2\,|\,2^u) &= 4u-1 \quad \textit{for } u \ge 1, \\ M_A(3\,|\,2^u) &= 6u-5 \quad \textit{for } u \ge 2, \\ M_A(4\,|\,2^u) &= 8u-9 \quad \textit{for } u \ge 2. \end{aligned}$$

Proof. We prove the lemma by induction on u. It is easy to check each initiation. For induction we employ formulae (3), (4), and (5) to yield the following:

$$\begin{aligned} M_A(1\,|\,2^u) &= 1 + M_A(1\,|\,2^{u-1}) + M_A(0\,|\,2^{u-1}) \\ &= 1 + 2(u-1) + 1 + 1 \\ &= 2u+1, \\ M_A(2\,|\,2^u) &= \max(1 + M_A(2\,|\,2^{u-1}, 1 + 2M_A(1\,|\,2^{u-1}))) \\ &= \max(4u-4, 4u-1) \\ &= 4u-1, \\ M_A(3\,|\,2^u) &= \max(1 + M_A(3\,|\,2^{u-1}), 1 + M_A(1\,|\,2^{u-1}) + M_A(2\,|\,2^{u-1})) \\ &= \max(6u-10, 6u-5) \\ &= 6u-5, \\ M_A(4\,|\,2^u) &= \max_{0\le d'\le 4}(M_A(d'\,|\,2^{u-1}) + M_A(4-d'\,|\,2^{u-1})) \\ &= \max(8u-17, 8u-11, 8u-9) \\ &= 8u-9. \qquad \square \end{aligned}$$

LEMMA 3.8. *Let* $u+1 = \lceil \log n \rceil$, $v = \lceil \log(n-2^u) \rceil$, *and* $w = \lceil \log(n - 2^u - 2^{v-1}) \lceil$. *Then*

$$\begin{aligned} &M_A(1\,|\,n) \le 2(u+1)+1, \\ &M_A(2\,|\,n) \le \max(4u+1, 2(u+v+1)+1), \\ &M_A(3\,|\,n) \le \max(6u-3, 4u+2v+1), \\ &M_A(4\,|\,n) \le \max(8u-7, 6u+2v-3, 4u+4v-3, 4u+2v+2w+1). \end{aligned}$$

Proof. Use the recursive formula (3), and note that $M_A(d\,|\,n-2^u) \le M_A(d\,|\,2^v)$. □

Now, we prove Theorem 3.1 in the case of $1 \le d \le 4$ as follows. Note that u and v are defined the same as in Lemma 3.8.

For $d = 1$, $M_A(1\,|\,n) \le 2(u+1)+1 = 2M(1,n)+1$.

For $d = 2$, if $v < u$, then $M_A(2\,|\,n) \le 4u+1$ and

$$M(2,n) \ge \left\lceil \log \binom{n}{2} \right\rceil \ge \left\lceil \log \frac{(2^u+1)2^u}{2} \right\rceil = 2u.$$

So $M_A(2\,|\,n) \le 2M(2,n)$. If $u = v$, then $M_A(2\,|\,n) = 2(2u+1)+1$ and

$$\begin{aligned} M(2,n) &\ge \left\lceil \log \binom{n}{2} \right\rceil \\ &\ge \left\lceil \log \frac{(2^u+2^{u-1}+1)(2^u+2^{u-1})}{2} \right\rceil \\ &\ge \lceil \log(2^{2u}+2^{2u-3}) \rceil = 2u+1. \end{aligned}$$

Thus (1) holds.

For $d = 3$ verification is trivial for $u = 1$. Next, we consider $u \ge 2$. If $v < u$, then $M_A(3\,|\,n) \le 2(3u-2)+1$ and

$$\begin{aligned} M(3,n) &\ge \left\lceil \log \binom{n}{3} \right\rceil \\ &\ge \left\lceil \log \frac{(2^u+1)2^u(2^u-1)}{6} \right\rceil \\ &\ge \lceil \log((2^u+1)2^{2u-3}) \rceil \\ &\ge 3u-2. \end{aligned}$$

Thus (1) holds. If $u = v$, then $M_A(3\,|\,n) \le 6u+1$ and

$$\begin{aligned} M(3,n) &\ge \left\lceil \log \binom{n}{3} \right\rceil \\ &\ge \left\lceil \log \frac{(2^u+2^{u-1}+1)(2^u+2^{u-1})(2^u+2^{u-1}-1)}{6} \right\rceil \\ &\ge \lceil \log((2^{2u+1}+2^{2u-2}-1)2^{u-2}) \rceil \\ &\ge 3u. \end{aligned}$$

For $d = 4$ it is trivial to verify (1) for $u = 1$ and 2. Next, consider $u \ge 3$. If $v < u$ and $w < u-1$, then $M_A(4\,|\,n) \le 2(4u-3)+1$ and

$$\begin{aligned} \left\lceil \log \binom{n}{4} \right\rceil &\ge \left\lceil \log \frac{(2^u+1)2^u(2^u-1)(2^u-2)}{8 \cdot 3} \right\rceil \\ &= \left\lceil \log \frac{(2^{3u}-2^{2u}-2^{u-1}+1)2^{u-2}}{3} \right\rceil \\ &\ge 4u-3. \end{aligned}$$

Thus (1) holds. If $u = v$ and $w < u - 1$, then $M_A(4 \,|\, n) \le 2(4u - 2) + 1$ and

$$\begin{aligned}
\left\lceil \log \binom{n}{4} \right\rceil &\ge \left\lceil \log \frac{(2^u + 2^{u-1} + 1)(2^u + 2^{u-1})(2^u + 2^{u-1} - 1)(2^u + 2^{u-1} - 2)}{8 \cdot 3} \right\rceil \\
&\ge \left\lceil \log \frac{((2^u + 2^{u-1})^2 - 1)((2^u + 2^{u-1} - 1)^2 - 1)}{3 \cdot 8} \right\rceil \\
&\ge \left\lceil \log \left(\frac{2^{2u+1} + 2^{2u-2} - 1}{3} \cdot \frac{2^{2u+1} + 2^{2u-2} - 2^u - 2^{u-1}}{8} \right) \right\rceil \\
&\ge \left\lceil \log((2^{2u-1} + 1)2^{2u-2}) \right\rceil \\
&\ge 4u - 2.
\end{aligned}$$

So (1) holds. If $w = u - 1$, then we must have $u = v$ and $M_A(4 \,|\, n) \le 2(4u - 1) + 1$. Note that $n \ge 2^u + 2^{u-1} + 2^{u-2} + 1$. Thus

$$\begin{aligned}
\left\lceil \log \binom{n}{4} \right\rceil &\ge \left\lceil \log \frac{((2^u + 2^{u-1} + 2^{u-2})^2 - 1)((2^u + 2^{u-1} + 2^{u-2} - 1)^2 - 1)}{3 \cdot 8} \right\rceil \\
&\ge \left\lceil \log \left(\frac{2^{2u+1} + 2^{2u} + 2^{2u-4} - 1}{3} \cdot \frac{2^{2u+1} + 2^{2u} + 2^{2u-4} - 2^u - 2^{u-1} - 2^{u-2}}{8} \right) \right\rceil \\
&\ge \left\lceil \log((2^{2u} + 1)2^{2u-2}) \right\rceil \\
&\ge 4u - 1.
\end{aligned}$$

Therefore, (1) holds.

Appendix 2: Proof of Theorem 3.1 for $5 \le d \le 40$. To save some computation, let us first compute $n_d = \min\{n \mid f(n, d) > 1\}$ for $5 \le d \le 40$, where $f(n, d)$ is the function defined by (2). The result is shown in the following.

$$\begin{array}{llllll}
n_5 = 73, & n_6 = 44 & n_7 = 38, & n_8 = 37, & n_9 = 37, & n_{10} = 37, \\
n_{11} = 39, & n_{12} = 40, & n_{13} = 42, & n_{14} = 43, & n_{15} = 45, & n_{16} = 47, \\
n_{17} = 49, & n_{18} = 50, & n_{19} = 52, & n_{20} = 54, & n_{21} = 56, & n_{22} = 58, \\
n_{23} = 60, & n_{24} = 62, & n_{25} = 64, & n_{26} = 66, & n_{27} = 68, & n_{28} = 70, \\
n_{29} = 72, & n_{30} = 74, & n_{31} = 76, & n_{32} = 78, & n_{33} = 80, & n_{34} = 82, \\
n_{35} = 84, & n_{36} = 86, & n_{37} = 88, & n_{38} = 90, & n_{39} = 92, & n_{40} = 94.
\end{array}$$

From the preceding equations we see that for $21 \le d \le 40$, $d/n_d \ge 8/21$. Therefore, for $21 \le d \le 40$, (1) holds. Now, for $5 \le d \le 20$ and $(21/8)d < n < n_d$ we compute $M_a(d \,|\, n)$ and $\ell(d, n)$. The results are given in Table 1 and show that $M_A(d \,|\, n) \le 2\ell(d, n) + 1$.

TABLE 1
$(d, n, l(d, n), M_A(d \mid n))$ *values for* $5 \leq d \leq 20$ *and* $(21/8)d < d < n_d$.

(5, 14, 12, 23)	(5, 15, 13, 25)	(5, 16, 13, 25)	(5, 17, 13, 27)	(5, 18, 14, 27)	(5, 19, 14, 29)
(5, 20, 14, 29)	(5, 21, 15, 31)	(5, 22, 15, 31)	(5, 23, 16, 31)	(5, 24, 16, 31)	(5, 25, 16, 33)
(5, 26, 17, 33)	(5, 27, 17, 33)	(5, 28, 17, 33)	(5, 29, 17, 35)	(5, 30, 18, 35)	(5, 31, 18, 35)
(5, 32, 18, 35)	(5, 33, 18, 37)	(5, 34, 19, 37)	(5, 35, 19, 37)	(5, 36, 19, 37)	(5, 37, 19, 39)
(5, 38, 19, 39)	(5, 39, 20, 39)	(5, 40, 20, 39)	(5, 41, 20, 41)	(5, 42, 20, 41)	(5, 43, 20, 41)
(5, 44, 21, 41)	(5, 45, 21, 41)	(5, 46, 21, 41)	(5, 47, 21, 41)	(5, 48, 21, 41)	(5, 49, 21, 43)
(5, 50, 22, 43)	(5, 51, 22, 43)	(5, 52, 22, 43)	(5, 53, 22, 43)	(5, 54, 22, 43)	(5, 55, 22, 43)
(5, 56, 22, 43)	(5, 57, 22, 45)	(5, 58, 23, 45)	(5, 59, 23, 45)	(5, 60, 23, 45)	(5, 61, 23, 45)
(5, 62, 23, 45)	(5, 63, 23, 45)	(5, 64, 23, 45)	(5, 65, 23, 47)	(5, 66, 24, 47)	(5, 67, 24, 47)
(5, 68, 24, 47)	(5, 69, 24, 47)	(5, 70, 24, 47)	(5, 71, 24, 47)	(5, 72, 24, 47)	(6, 16, 15, 27)
(6, 17, 15, 29)	(6, 18, 15, 29)	(6, 19, 16, 31)	(6, 20, 16, 31)	(6, 21, 16, 33)	(6, 22, 17, 33)
(6, 23, 17, 35)	(6, 24, 18, 35)	(6, 25, 18, 37)	(6, 26, 18, 37)	(6, 27, 19, 37)	(6, 28, 19, 37)
(6, 29, 19, 39)	(6, 30, 20, 39)	(6, 31, 20, 39)	(6, 32, 20, 39)	(6, 33, 21, 41)	(6, 34, 21, 41)
(6, 35, 21, 41)	(6, 36, 21, 41)	(6, 37, 22, 43)	(6, 38, 22, 43)	(6, 39, 22, 43)	(6, 40, 22, 43)
(6, 41, 23, 45)	(6, 42, 23, 45)	(6, 43, 23, 45)	(7, 19, 17, 33)	(7, 20, 18, 33)	(7, 21, 18, 35)
(7, 22, 18, 35)	(7, 23, 19, 37)	(7, 24, 19, 37)	(7, 25, 20, 39)	(7, 26, 20, 39)	(7, 27, 20, 41)
(7, 28, 21, 41)	(7, 29, 21, 43)	(7, 30, 21, 43)	(7, 31, 22, 43)	(7, 32, 22, 43)	(7, 33, 23, 45)
(7, 34, 23, 45)	(7, 35, 23, 45)	(7, 36, 23, 45)	(7, 37, 24, 47)	(8, 22, 20, 37)	(8, 23, 20, 39)
(8, 24, 21, 39)	(8, 25, 21, 41)	(8, 26, 22, 41)	(8, 27, 22, 43)	(8, 28, 22, 43)	(8, 29, 23, 45)
(8, 30, 23, 45)	(8, 31, 23, 47)	(8, 32, 24, 47)	(8, 33, 24, 49)	(8, 34, 25, 49)	(8, 35, 25, 49)
(8, 36, 25, 49)	(9, 24, 22, 41)	(9, 25, 23, 43)	(9, 26, 23, 43)	(9, 27, 24, 45)	(9, 28, 24, 45)
(9, 29, 24, 47)	(9, 30, 25, 47)	(9, 31, 25, 49)	(9, 32, 25, 49)	(9, 33, 26, 51)	(9, 34, 26, 51)
(9, 35, 27, 53)	(9, 36, 27, 53)	(10, 27, 25, 47)	(10, 28, 26, 47)	(10, 29, 26, 49)	(10, 30, 26, 49)
(10, 31, 27, 51)	(10, 32, 27, 51)	(10, 33, 27, 53)	(10, 34, 28, 53)	(10, 35, 28, 55)	(10, 36, 29, 55)
(11, 29, 28, 51)	(11, 30, 28, 51)	(11, 31, 28, 53)	(11, 32, 29, 53)	(11, 33, 29, 55)	(11, 34, 29, 55)
(11, 35, 30, 57)	(11, 36, 30, 57)	(11, 37, 31, 59)	(11, 38, 31, 59)	(12, 32, 30, 55)	(12, 33, 31, 57)
(12, 34, 31, 57)	(12, 35, 31, 59)	(12, 36, 32, 59)	(12, 37, 32, 61)	(12, 38, 33, 61)	(12, 39, 33, 63)
(13, 35, 33, 61)	(13, 36, 33, 61)	(13, 37, 34, 63)	(13, 38, 34, 63)	(13, 39, 35, 65)	(13, 40, 35, 65)
(13, 41, 35, 67)	(14, 37, 35, 65)	(14, 38, 36, 65)	(14, 39, 36, 67)	(14, 30, 27, 67)	(14, 41, 37, 69)
(14, 42, 37, 69)	(15, 40, 38, 69)	(15, 41, 39, 71)	(15, 42, 39, 71)	(15, 43, 39, 73)	(15, 44, 40, 73)
(16, 43, 41, 75)	(16, 44, 41, 75)	(16, 45, 42, 77)	(16, 46, 42, 77)	(17, 45, 43, 79)	(17, 46, 44, 79)
(17, 47, 44, 81)	(17, 48, 45, 81)	(18, 48, 46, 83)	(18, 49, 47, 85)	(19, 50, 49, 87)	(19, 51, 49, 89)
(20, 53, 51, 93)					

REFERENCES

[1] R. Ahlswede and I. Wegener, *Search Problems*, John Wiley, New York, 1987.

[2] M. Aigner, *Combinatorial Search*, John Wiley & Sons, New York, 1988.

[3] J. A. Aslam and A. Dhagat, *Searching in the presence of linearly bounded errors*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1991, pp. 486–493.

[4] A. Bar-Noy, F. K. Hwang, I. Kessler, and S. Kutten, *A New Competitive Algorithm for Group Testing*, manuscript.

[5] R. Dorfman, *The detection of defective members in a large population*, Ann. Math. Statist., 14 (1943), pp. 4436–4440.

[6] D. Z. Du and F. K. Hwang, *Minimizing a combinatorial function*, SIAM J. Alg. Discrete Math., 3 (1982), pp. 523–528.

[7] D. Z. Du and F. K. Hwang, *Competitive group testing*, in On-Line Algorithms, L. A. McGeoch and D. D. Sleator, eds., Association for Computing Machinery, New York, 1992, pp. 125–134.

[8] P. Erdos and A. Renyi, *On two problems of information theory*, Publ. Hung. Acad. Sci., 8 (1964), pp. 241–254.

[9] M. C. Hu, F. K. Hwang, and J. K. Wang, *A boundary problem for group testing*, SIAM J. Alg. Discrete Math., 2 (1981), pp. 81–87.

[10] R. M. Karp, E. Upfal, and A. Wigderson, *Are search and decision problems computationally equivalent*? in Proc. 17th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1985, pp. 464–475.

[11] K.-I. Ko, *Searching for two objects by underweight feedback*, SIAM J. Discrete Math., 1 (1988), pp. 65–70.

[12] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for on-line problems*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 322–333.

[13] E. M. Palmer, *Graphical Evolution: An Introduction to the Theory of Random Graphs*, John Wiley & Sons, New York, 1985.

[14] R. L. Rivest, A. R. Meyer, D. J. Kleitman, K. Winklmann, and J. Spencer, *Coping with errors in binary search procedures*, J. Comput. System Sci., 20 (1980), pp. 396–404.

[15] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

[16] M. Sobel and P. A. Groll, *Group-testing to eliminate efficiently all defectives in a binominal sample*, Bell Syst. Tech. J., 38 (1959), pp. 1179–1252.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 97–119, February 1994

THE COMPLEXITY OF DECISION VERSUS SEARCH*

MIHIR BELLARE[†] AND SHAFI GOLDWASSER[‡]

Abstract. A basic question about NP is whether or not search reduces in polynomial time to decision. This paper indicates that the answer is negative: Under a complexity assumption (that deterministic and nondeterministic double-exponential time are unequal) a language in NP for which search does not reduce to decision is constructed.

These ideas extend in a natural way to interactive proofs and program checking. Under similar assumptions, the authors present languages in NP for which it is harder to prove membership interactively than it is to decide this membership, and languages in NP that are not checkable.

Key words. NP-completeness, self-reducibility, interactive proofs, program checking, sparse sets, quadratic residuosity

AMS subject classifications. 68Q15, 03D15, 94A60

1. Introduction.

The work on interactive proofs brought back to light a basic question: How powerful does a prover need to be to convince a verifier of membership in a language L? Clearly, the prover needs at least the power to decide the language for himself. The question we focus on is whether this is enough.

There are interactive proofs known for complete problems in NP, $P^{\#P}$, and PSPACE where it is sufficient for the prover to be able to decide membership in the language. Such power is also sufficient for almost all of the languages in IP that have been closely examined (specifically, the languages of graph isomorphism, graph nonisomorphism [GMW], and quadratic nonresiduosity [GMR]). On the other hand, all known interactive proofs for complete languages for coNP require the prover to do more than decide membership in the language. Similarly, all known interactive proofs for the language of quadratic residuosity require the prover to do more than decide quadratic residuosity.

As we will see, this is essentially a generalization of the old question of whether search problems reduce to their decision counterpart for NP. Namely is computing a witness for membership in $L \in$ NP any harder than establishing the existence of such a witness? For NP-complete problems it is well known that the answer is no: given an oracle for membership a witness can be computed in polynomial time. But for general $L \in$ NP the problem remains open.

In this paper we use natural complexity assumptions to indicate that proving membership may be harder than deciding it. As a first example we look at decision versus search in NP. We then turn to interactive proofs and finally apply the same ideas to derive results on the difficulty of program checking.

Let us proceed to describe our results in detail.

1.1. Decision versus search in NP.

Before we can present our results, we need to say what we mean by "search," "decision," and the "reduction" of the former to the latter. We will

*Received by the editors March 23, 1992; accepted for publication (in revised form) August 21, 1992. These results appeared in preliminary form in [BG]. Later, merged with [BF], they appeared in [BBFG].

[†] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (mihir @watson.ibm.com). Part of the research of this author was performed at the Massachusetts Institute of Technology. This author was partially supported by the National Science Foundation grant CCR-8719689 and Defense Advanced Research Projects Agency grant N00014-89-J-1988.

[‡] Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139 (shafi@theory.lcs.mit.edu). This author was partially supported by National Science Foundation grant 25801, Defense Advanced Research Projects Agency grant 71949, the Princeton University Department of Computer Science, and U.S.–Israel Binational Science Foundation grant 86-00301.

keep the discussion here informal; for formal definitions we refer the reader to §2.1. We start with some terminology.

Suppose $\rho(\cdot, \cdot)$ is a polynomial-time computable binary relation. We let $\rho(x) = \{w : \rho(x, w) = 1\}$ be the set of all ρ-witnesses for x. We say that ρ is an NP-relation if there exists a constant c such that for all $x \in \{0, 1\}^*$ it is the case that $\rho(x) \subseteq \{0, 1\}^{|x|^c}$. We let $L_\rho = \{x \in \{0, 1\}^* : \rho(x) \neq \emptyset\}$.

Now let $L \subseteq \{0, 1\}^*$ be a language. We say that ρ defines L if $L = L_\rho$. Clearly, $L \in$ NP if and only if there exists an NP-relation that defines L. It is important to note, however, that for any particular NP language L, there are many (different) NP-relations that define it.

Associated to any NP language L is a (single) decision problem and a class of search problems. The decision problem is, of course, just the problem of deciding membership in L. As for the search problems, there is one for each NP-relation that defines L, and the search problem corresponding to a particular NP-relation ρ that defines L is the following: given $x \in L$, find a ρ-witness for x. For example, if L is SAT, then the decision problem is to decide whether or not a given formula is satisfiable. One of the associated search problems is to determine a truth assignment of a given satisfiable formula (but there are other associated search problems as well).

We are interested in defining what it means for search to reduce to decision for L. As a means for obtaining the definition and understanding the issues involved, we begin by discussing a less general notion: that of reducing search to decision for an NP-relation ρ (defining L).

Fix a particular NP-relation ρ that defines L. We say that "search reduces to decision for ρ" if the search problem for ρ is solvable in polynomial time, given an oracle for the decision of $L = L_\rho$. More precisely, search reduces to decision for ρ if there exists a polynomial-time oracle machine W such that for all $x \in L$ it is the case that $W^L(x)$ (the output of W with oracle L and input x) is a ρ-witness for x. Intuitively, the search problem for ρ is no harder than the decision problem for the corresponding language.

We are now ready to state what it means for search to reduce to decision for a *language* $L \in$ NP. We recall that there are many different NP-relations defining L. In general, search might reduce to decision for some of these and not for others. Our definition is to say that search reduces to decision for L as long as there is *some* NP-relation ρ (defining L) such that search reduces to decision for ρ. In other words, we say that search reduces to decision for L as long as at least one of the (many different) search problems associated to the decision problem for L is no harder than this decision problem.

The motivation for this definition, which stems from the question of whether proving membership can be harder than deciding it, will become clearer as we go on. For the moment, it is more important to stress the generality of our definition and the strength of negative conclusions that are based on it. In particular, to say that search does *not* reduce to decision for a particular language L (as in the conclusion of the theorem that follows) is to make a strong statement indeed, because it means that for *all* ρ defining L it is the case that search does not reduce to decision for ρ. That is, *all* the search problems corresponding to the decision problem of L are harder than this decision problem. In particular, the existence of a language for which search does not reduce to decision certainly implies the existence of an NP-relation for which search does not reduce to decision.

To state the theorem we first need the following definitions:

$$\mathrm{EE} = \bigcup_{c \geq 0} \mathrm{DTIME}(2^{c2^n}) \quad \text{and} \quad \mathrm{NEE} = \bigcup_{c \geq 0} \mathrm{NTIME}(2^{c2^n}).$$

THEOREM 1.1. *Suppose EE $\neq$ NEE. Then there is a language in* NP *for which search does not reduce to decision.*

Note that the conclusion (of Theorem 1.1) implies P $\neq$ NP. Whether the assumption could be reduced to P $\neq$ NP (or even E $\neq$ NE) remains an open question.

We note that if L is NP-complete, then for *any* NP-relation ρ that defines L, it is the case that search reduces to decision for ρ (a consequence of the self-reducibility and NP-completeness of SAT as well as certain features of the proof of Cook's theorem [Co]; this is one of the most basic and well-known facts in the theory of computation). In particular, by our definition search certainly reduces to decision for any NP-complete language. So the hard problems (from the point of view of search versus decision) will necessarily be non-NP-complete. In particular, the language of the conclusion of Theorem 1.1 is not NP-complete.

The decision-versus-search question has attracted the attention of researchers ever since NP was introduced (we survey some of the work on this subject in §1.5). However, we note that previous work has focused on the question of whether search reduces to decision for NP-relations (not NP-languages), and the conclusions have been weaker than ours.

1.2. Competitive proof systems: The natural extension. NP represents the simplest kind of proof system. An NP proof system for L is defined by a polynomial time verifier V. This verifier talks to a prover who, on an input x common to both parties, is allowed to send the verifier a single message of length polynomial in $n = |x|$. As a function of this message and the common input, the verifier decides whether or not to accept (this decision of the verifier is a polynomial-time binary predicate ρ evaluated on the common input and the prover's message). In the case that $x \in L$ there must exist some *deterministic* prover P who can convince the verifier to accept (this is the *completeness* condition). In the case that $x \notin L$ no prover should be able to convince the verifier to accept (this is the *soundness* condition). We usually specify an NP proof system by a pair (P, V), where P is a prover satisfying the completeness condition. Clearly, $L \in$ NP if and only if it possesses an NP proof system.

How powerful need the prover P be in an NP proof system for L? It is clear the prover must have at least the ability to decide L for himself. Let us call an NP proof system (P, V) *competitive* if this minimal ability is also sufficient; more precisely, (P, V) is competitive if P runs in polynomial time given an oracle for L. It now becomes clear that the question of whether or not search reduces to decision for $L \in$ NP captures the computational difficulty of the prover's task under this competitive measure of complexity. More precisely, we observe that L has a competitive NP proof system if and only if search reduces to decision for L. Thus Theorem 1.1 indicates that there is a language $L \in$ NP to give an NP proof of which any prover must use power over and above that necessary to decide L.

NP proof systems, however, are very restrictive. It becomes natural to ask, would the prover's task be alleviated if the parties were allowed interaction and the proof was now only required to be correct with high probability? In other words, we now consider interactive proofs (see [GMR]). We recall that in an interactive proof both parties are allowed to be probabilistic and the parties are allowed to exchange messages, for a polynomial number of rounds, before the verifier decides whether or not to accept. Completeness and soundness are required to hold only with high probability (see §4.1 for precise definitions). Let us call an interactive proof system competitive if the prover[1] runs in probabilistic polynomial time, given access to L as an oracle. Then does every language in NP (and more generally in IP) have a competitive interactive proof? In other words, does the extra leeway provided by interaction and randomness reduce the burden on the prover, or does the discrepancy between proving and deciding remain even if coins and interaction are allowed?

Quadratic residuosity provides a telling example. Let QR $= \{(x, N) : \exists y \in Z_N^* \text{ s.t. } x \equiv y^2 (\text{mod } N)\}$, and let QNR $= \{(x, N) : \neg\exists y \in Z_N^* \text{ s.t. } x \equiv y^2 (\text{mod} N)\}$. Search is not known

[1] The prover here refers, of course, to the "honest" prover of the completeness condition; the soundness condition of the proof system is, as usual, required to hold with respect to any (computationally unbounded) prover.

to reduce the decision for QNR; in all known NP proof systems for QNR the prover requires the ability to factor N, and factoring is not known to be reducible to quadratic residuosity. Yet, we do know of *interactive* proofs for membership in QNR where it suffices for the prover to be able to tell membership in QR (i.e., QNR does have competitive interactive proofs) [GMR]. On the other hand, there is no known interactive proof for QR where it suffices for the prover to be able to decide membership in QR (i.e., QR is *not* known to have a competitive interactive proof).

Our next result indicates that, in general, interaction and randomness will not make the prover's task easier. More precisely, we indicate that not all languages in NP have competitive interactive proofs. Letting BPEE denote the class of languages recognized with bounded error by a probabilistic TM running in time 2^{c2^n} for some constant $c \geq 0$, we have the following theorem.

THEOREM 1.2. *If* NEE $\not\subseteq$ BPEE, *then there is a language in* NP *that does not have a competitive interactive proof.*

The complexity of a prover in an interactive proof system is a basic question that is attracting a fair amount of attention (see §1.5). The notion of competitive interactive proofs that we introduce provides an new angle from which to understand this question; whereas past work has focused on providing upper bounds on the complexity of provers, we are instead trying to understand the comparative complexity of proving versus deciding.

1.3. Program checking. We briefly mention our results on program checking that are in the same vein as the preceding.

Blum and Kannan [BK] introduced the notion of program checkers (see §5 for full definitions). Negative results in this domain begin with Yao [Ya], who presented a language in deterministic space $2^{n^{\log\log n}}$ that does not have a checker. Beigel and Feigenbaum [BF] and Krawczyk [Kr] improved this to deterministic space $n^{\log^* n}$. The question of whether there are languages of reasonable complexity that are not checkable was answered by Beigel and Feigenbaum [BF] under an assumption: they showed there was one such in NP provided that nondeterministic *triple* exponential time is not contained in bounded probabilistic *triple* exponential time. We improve the assumption to *double* exponential time. Namely, we have the following.

THEOREM 1.3. *If* NEE $\not\subseteq$ BPEE, *then there is a language in* NP *that does not have a checker.*

1.4. A natural candidate? Clearly, it would be most interesting to exhibit an example of a *natural* problem in NP for which there are no competitive interactive proofs. A candidate example—as we indicated in §1.2—is the quadratic residuosity problem. Let us consider the interactive proofs known for membership both in QR and QNR in more detail.

First, for membership of (x, N) in QNR, the following protocol is repeated k times (see [GMR]). The verifier tosses a coin $c \in \{0, 1\}$. If $c = 1$, then the verifier sends the prover $z = xr^2 \bmod N$ for random $r \in Z_N^*$; else the verifier sends the prover $z = r^2 \bmod N$ for random $r \in Z_N^*$. The prover is then asked to guess the value of c. If the prover guesses correctly in each repetition of the protocol, then the verifier accepts. Clearly, if $(x, N) \in$ QNR, then $c = 1$ if and only if $(z, N) \notin$ QR. Thus it is sufficient for the prover to be able to determine membership in QR in order to guess c, and the prover is competitive. On the other hand, if $(x, N) \notin$ QNR, then the probability that the verifier accepts is no greater than 2^{-k}.

How about proving that $(x, N) \in$ QR? A simple proof would be the factorization of N or a $y \in Z_N^*$ such that $x \equiv y^2 \pmod N$. In fact, all known interactive proofs of this fact require the ability to factor N. Because we do not know whether factoring reduces to deciding quadratic residuosity, it remains an intriguing open problem whether membership in QR can be interactively proved by a probabilistic polynomial-time prover with access to a QR oracle. In particular, if the answer to this question were negative, we would get the following interesting number-theoretic implication: integer factorization is not polynomial-time reducible to deciding quadratic residuosity.

We note that there are special classes of integers N for which $(x, N) \in \mathrm{QR}$ can be interactively proved be a probabilistic polynomial-time prover with access to a QR oracle. For example, this can be done when N is the product of a constant number of primes (see §6).

1.5. Related work. We discuss related work on decision versus search and the complexity of provers.

1.5.1. Decision versus search for NP. The decision-versus-search question has attracted the attention of researchers ever since NP was introduced. It has been studied in many different contexts and from many angles, and, in particular, many results have indicated that for NP, search is likely to be harder than decision. We stress that all these results are about search versus decision for NP-relations, not NP-languages. So, in that sense, the conclusions are weaker than ours.

Let us now describe some of this work. In what follows we let ρ denote a polynomial-time computable binary predicate.

Valiant [Va] appears to have been the first to indicate that there are NP-relations ρ for which search is unlikely to reduce to decision; specifically, assuming $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$, he presents a particular NP-relation ρ with the property that search does not reduce to decision for ρ. However, the underlying language L_ρ in Valiant's result is easy; in fact, it equals $\{0, 1\}^*$. Borodin and Demers [BD] strengthen Valiant's result in this regard by showing that under the same assumption there is an NP-relation ρ for which $L_\rho \in \mathrm{NP} - \mathrm{P}$, but search still does not reduce to decision for ρ. Hartmanis and Hemachandra [HH] present results similar to Valiant's but assume $\mathrm{P} \neq \mathrm{UP} \cap \mathrm{coUP}$.

Impagliazzo and Naor [IN] indicate that, at least in relativized worlds, the assumption $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ is not necessary for the conclusion of Valiant's result. More precisely, they present a relativized world in which $\mathrm{P} = \mathrm{NP} \cap \mathrm{coNP}$, but there exists an NP-relation ρ such that $L_\rho = \{0, 1\}^*$ and search does not reduce to decision for ρ.

The assumption $\mathrm{P} \neq \mathrm{NP}$ suffices to indicate that the usual method of self-reduction (in which one constructs a witness bit by bit, given an oracle for the language) may not always work: Selman [Se] shows that under this assumption there is an NP-relation ρ for which $L_\rho \in \mathrm{NP} - \mathrm{P}$, but, given a pair of strings (x, u) and an oracle for L_ρ, it is impossible to decide in polynomial time whether or not there is an extension of u that is a ρ-witness for x.

We stress again that none of the preceding work addresses the problem we consider. These other authors focus on NP-relations, asking whether there exist (specific) NP-relations ρ for which it is impossible to reduce the search problem for ρ to the corresponding decision problem for L_ρ. We focus on NP languages, asking whether there exists a language L for which, for *any* associated search problem ρ, it is impossible to reduce the search problem for ρ to its corresponding decision.

1.5.2. Decision versus search in other settings. The usual reduction of search to decision has a strong sequential flavor, and Karp, Upfal, and Wigderson [KUW] investigate the degree to which this is necessary. Ben-David, Chor, Goldreich, and Luby [BCGL] investigate the decision-versus-search question in the context of the average-case complexity. Impagliazzo and Tardos [IT] consider the decision-versus-search question in the exponential case and present an oracle relative to which $\mathrm{E} = \mathrm{NE}$, but there is an exponential-time binary predicate whose search problem is not solvable in exponential time.

1.5.3. The complexity of provers. Several recent works present results on the complexity of provers in interactive proofs. Let us describe some of them.

Shamir's [Sh] result implies that polynomial-space provers suffice to prove PSPACE languages. The best upper bound on the complexity of a prover of a coNP language, due to Lund, Fortnow, Karloff, and Nisan [LFKN], is probabilistic, polynomial time with a #P oracle.

Bellare and Petrank [BP] investigate the complexity of zero-knowledge (ZK) provers and indicate that such provers can be reasonably efficient; specifically, they show that any language possessing a statistical ZK interactive proof possesses one with a prover that is a probabilistic, polynomial-time machine with access to an NP oracle.

In the case of multiple-prover proofs, Babai, Fortnow, and Lund [BFL] show that exponential-time provers suffice for exponential-time languages.

We stress that all these works are concerned with upper bounding the complexity of provers in an absolute sense. The model of competitive interactive proofs that we introduce here is for the purpose of studying the complexity of provers in a different way, namely, in terms of the comparative complexity of proving versus deciding.

1.5.4. Recent work. Independently of this work, Impagliazzo and Sudan [IS] show that if NE $\neq$ coNE, then there is a language in NP for which search does not reduce to decision. Here the conclusion is the same as in Theorem 1.1, but the assumption is different (and not known to be either weaker or stronger). They also show that if E $\neq$ NE, then there is an NP-relation ρ for which search does not reduce to decision; this is the same conclusion as in the aforementioned result of Borodin and Demers [BD] but under an assumption that is different from that of [BD] (but, again, not known to be either weaker or stronger). Finally, Spielman [Sp] has constructed an uncheckable set in $\sum_2^{\mathrm{P}}$ under the assumption that $\sum_2^{\mathrm{E}} \neq \Pi_2^{\mathrm{E}}$.

1.6. Relations to other notions. We focus in this paper on (competitive) interactive proofs and checking. Related notions are function-restricted interactive proofs [BK], multiple-prover interactive proofs [BGKW], and coherence [Ya]. Here we discuss how these notions relate to ours and also how our results impinge on them. First, let us list (the complexity classes corresponding to) the notions in this area.

1.6.1. The complexity classes in this area. The following are the (main) complexity classes on which the ensuing discussion will focus.

$$\begin{aligned}
\mathrm{compNP} &= \{L : L \text{ has a competitive NP-proof system}\}\\
&= \{L \in \mathrm{NP} : \text{ search reduces to decision for } L\},\\
\mathrm{IP} &= \{L : L \text{ has an interactive proof system}\},\\
\mathrm{compIP} &= \{L : L \text{ has a competitive interactive proof system}\},\\
\mathrm{frIP} &= \{L : L \text{ has a function-restricted interactive proof system}\},\\
\mathrm{MIP} &= \{L : L \text{ has a multi-prover interactive proof system}\},\\
\mathrm{Check} &= \{L : L \text{ is checkable}\},\\
\mathrm{Coh} &= \{L : L \text{ is coherent}\}.
\end{aligned}$$

1.6.2. Function-restricted interactive proofs. Function-restricted interactive proof systems are a variant of interactive proof systems introduced by Blum and Kannan [BK]. Like competitive interactive proofs, they make the restriction that the honest prover be a probabilistic, polynomial-time machine with access to an oracle for the language in question. But, in contrast to competitive interactive proofs, they also restrict the dishonest prover. Specifically, they ask that a dishonest prover be a function from verifier messages to strings. In particular, the response of the dishonest prover to a verifier message is not allowed to depend on previous questions of the verifier (that is, its messages are independent of the history).

As we will see (Lemma 4.3) it is the case that comIP $\subseteq$ frIP. Blum and Kannan also established that Check $\subseteq$ frIP. On the other hand, on the basis of the techniques of [FRS] one can show that frIP $\subseteq$ MIP.

Function-restricted interactive proofs were introduced in order to relate program checking to interactive proofs. We introduce competitive interactive proofs to address the question of how much power is necessary for the honest prover to prove membership interactively; whence it is imperative to not weaken the definition of interactive proofs by making assumptions on the power of the dishonest prover. However, we use the notion of function-restricted proof systems to provide a unified treatment of our results. We establish the main technical lemmas needed for our proofs in terms of the equivalent notion of "deciders" (see §3) and then use these lemmas to derive our results on competitive interactive proofs and checking in a simple way. In particular (improving [BF]; see the following), we do show that if NEE $\not\subseteq$ BPEE, then there is a language in NP $-$ frIP.

1.6.3. Competitive multiple-prover proofs. One could define competitive multi-prover proofs. However, using techniques of [FRS], one can show that the corresponding class of languages is identical to frIP.

1.6.4. Coherence. The notion of coherence was introduced by Yao [Ya]. Informally, a language L is *coherent* if the membership of x in L can be decided in probabilistic polynomial time and bounded error by a machine (called the "examiner") that has access to L as an oracle but is allowed to query this oracle only on points different from x. If L is not coherent, we say it is incoherent. Beigel and Feigenbaum [BF] prove the existence of incoherent languages in NP under the assumption that nondeterministic *triple* exponential time is not contained in bounded probabilistic *triple* exponential time. Since checkable languages and languages in frIP are coherent (see [Ya], [BF]), they thereby establish the existence of uncheckable languages in NP, and languages in NP $-$ frIP, under the same assumption.

One can show that if search reduces to decision for L or if L has a competitive interactive proof, then also L is coherent. So the construction of incoherent sets yields negative results about these notions as well. However, the fact that our stronger results on all these notions (namely, decision versus search, competitive interactive proofs, checking, and function-restricted proofs) are obtained more directly (i.e., avoiding incoherence) indicates that coherence may not be the best approach to negative results in this area.

We note that, intuitively, coherence has a flavor different from that of the other notions we have considered; whereas the common underpinning of these others is the notion of a proof (interactive or noninteractive), coherence is not a form of proof. Indeed, the examiner gives no proof that $x \in L$—there is no guarantee as to what would happen if the examiner is run with an oracle different from L. It is by exploiting this prooflike quality of the notions we consider that we are able to derive results that are stronger than those derived by the coherence approach.

In this context we note also that one can separate the classes of checkable and coherent languages inside NP, assuming NEE $\not\subseteq$ BPEE (see Theorem 5.4).

1.6.5. Summary. We summarize relationships among the various complexity classes we have discussed. First, we introduce some notation. We define the triple-exponential time class

$$\mathrm{NEEE}^* = \bigcup_{c\geq 0} \mathrm{NTIME}(2^{2^{2^{n^c}}}).$$

Similarly, we let BPEEE* denote the class of languages recognized with bounded error by a probabilistic Turing machine (TM) running in time $2^{2^{2^{n^c}}}$ for some constant $c \geq 0$. The following inclusions are known or are easily derived from known techniques:

(1) compNP $\subseteq$ NP $\cap$ compIP.

(2) NP $\cup$ compIP $\subseteq$ IP $\subseteq$ MIP.

(3) compIP $\cup$ Check $\subseteq$ frIP $\subseteq$ MIP $\cap$ Coh.

Under the assumption NEE $\not\subseteq$ BPEE we establish the following:

(4) NP is not contained in any of the following: compNP, compIP, Check, frIP.

(5) NP $\cap$ Coh is not contained in any of the following: compNP, compIP, Check, frIP.

For the results of line (4), see Theorems 2.9, 4.4, 5.3, and 3.6. For those of line (5), see (Theorem 5.4 and) the discussion at the end of §5. Finally under the assumption NEEE* $\not\subseteq$ BPEEE* [BF], establish the following:

(6) NP $\not\subseteq$ Coh.

2. Decision versus search in NP. In this section, we present a simple construction of a language in NP for which search does not reduce to decision if it is assumed that EE $\neq$ NEE. In later sections we will extend the argument to interactive proofs and program checking. Let us begin with the definitions.

2.1. Definitions. The goal of this section is to make precise what we mean by "search reduces to decision for an NP language L." Since the issues were discussed at length in §1.1, we will here be brief, stating the (formal) definitions and limiting the discussion to essentials.

It is convenient to proceed in steps. We begin by defining NP-relations and saying what it means for search to reduce to decision for them. We then use this to say what it means for search to reduce to decision for an NP language.

DEFINITION 2.1. Let $\rho(\cdot, \cdot)$ be a polynomial-time computable binary relation, and let $x \in \{0, 1\}^*$. We let $\rho(x) = \{w \in \{0, 1\}^* : \rho(x, w) = 1\}$ and call the members of this set *ρ-witnesses* for x. We say that ρ is an *NP-relation* if there exists a constant $c \in \mathbf{N}$ such that for all $x \in \{0, 1\}^*$ it is the case that $\rho(x) \subseteq \{0, 1\}^{|x|^c}$. The *language defined by* ρ is $\{x \in \{0, 1\}^* : \rho(x) \neq \emptyset\}$ and is denoted L_ρ.

Note that if ρ is an NP-relation, then $L_\rho \in$ NP.

Notation. If $W(\cdot)$ is an oracle machine, then $W^L(x)$ denotes the output of W with oracle $L \subseteq \{0, 1\}^*$ and input $x \in \{0, 1\}^*$.

We now say what it means for search to reduce to decision for an NP-relation. An equivalent formulation of the definition that follows appears in [BD].

DEFINITION 2.2. Suppose ρ is an NP-relation and $W(\cdot)$ is a polynomial-time oracle machine. Let $L = L_\rho$. We say that W is a *ρ-witness finder* if for each $x \in L$ it is the case that $W^L(x) \in \rho(x)$. We say that *search reduces to decision for* ρ if there exists a ρ-witness finder.

Note that the witness finder is not restricted to any particular method (for example, it is not required that the length of queries be decreasing with time). Rather, any polynomial-time computation is allowed. This strengthens negative results.

We now wish to say what it means for search to reduce to decision for an NP-*language* (as opposed to an NP-relation). Begin with the following terminology.

DEFINITION 2.3. Suppose ρ is an NP-relation and $L \subseteq \{0, 1\}^*$ is a language. We say that *ρ defines L* if $L = L_\rho$.

Clearly, $L \in$ NP if and only if there exists an NP-relation that defines L. However, for any particular language $L \in$ NP there may be many different NP-relations that define L. If L is NP-complete, then search reduces to decision for *any* of these NP-relations. However, if L is not NP-complete, then search might reduce to decision for some of them but not for others.

In defining what it means for search to reduce to decision for L we have chosen to be liberal: we ask only that there be *some* NP-relation ρ defining L for which search reduces to decision.

DEFINITION 2.4. Suppose $L \subseteq \{0, 1\}^*$. We say that *search reduces to decision for* L if there exists an NP-relation ρ such that ρ defines L and search reduces to decision for ρ.

As we indicated in §1, our definition is motivated by interactive proofs and the question of whether proving membership is harder than deciding it. Proving membership in L is easy (in the sense that L has a competitive NP-proof system) as long as search reduces to decision for *some* NP-relation defining L, so that we are led to Definition 2.4. We note, in this context, that there are languages (such as $\{0, 1\}^*$) that are easy but that have an associated search problem that is hard [Va], and we certainly do not wish to think of search as being harder than decision for these languages. Appropriately, search does not reduce to decision for these languages according to our definition.

Finally, we note that the existence of a language for which search does not reduce to decision does, of course, imply the existence of an NP-relation for which search does not reduce to decision, so that negative results under our definition are stronger than those that simply conclude the existence of NP-relations for which search is harder than decision.

Whenever ρ is understood, we will say "witness" or "witness finder" rather than "ρ-witness" or "ρ-witness finder."

2.2. Uniformly log-sparse languages. Our proof will use languages that combine logarithmic sparseness with the property that it be possible to efficiently identify a logarithmic-sized superset of the strings below any given length. Let us proceed to the formal definitions.

DEFINITION 2.5. The *census function* $\mu_L : \mathbf{N} \to \mathbf{N}$ of $L \subseteq \{0, 1\}^*$ of a language L is defined by $\mu_L(n) = \sum_{i=0}^{n} |L \cap \{0, 1\}^i|$. We say that L is *log-sparse* if $\mu_L(n) = O(\log n)$.

That is, $\mu_L(n)$ is the number of strings in L that have length $\leq n$, and a language is log-sparse if it contains at most $O(\log n)$ strings of length at most n. Log-sparse languages are used in [HSI], where they are called "super-sparse."

The next definition formalizes the idea of being able to efficiently identify some superset of a language and then specifies the notion of uniform log-sparseness in which we are interested.

DEFINITION 2.6. We say that $C \subseteq \{0, 1\}^*$ is a *candidate selector* for L if C is polynomial-time decidable and $L \subseteq C$. We say that a language L is *uniformly log-sparse* if it has a log-sparse candidate selector.

As we will see in §2.3, the interest of uniformly log-sparse languages is that they form a class for which the problem of reducing search to decision is particularly hard. Let us end this section by stating a lemma that we will use later. This lemma generalizes work of Hartmanis, Sewelson, and Immerman [HSI], who show that there is a log-sparse language in NP − P if EE ≠ NEE.[2] Log-sparseness is weaker than uniform log-sparseness in that no candidate selector is required, but it is easy to see that a uniform log-sparseness language in NP − P nonetheless exists under the same assumption. For completeness we provide a sketch of the (entirely standard) proof.

LEMMA 2.7. *If* EE ≠ NEE, *then there is a uniformly log-sparse language in* NP − P.

Proof. We use a standard downward-separation argument. Assume EE ≠ NEE and suppose $L' \in$ NEE − EE. Define $L = \{y.0^{g(|y|)-|y|} : y \in L'\}$, where $g(k) = 2^{2^k}$. We claim that L is uniformly log-sparse and $L \in$ NP − P.

Define A to be the logarithm that on input $x \in \{0, 1\}^n$ behaves as follows. If n is not in the range of g, then A outputs 0. Else it computes $k = g^{-1}(n)$ and outputs 1 if and only if

[2] Hartmanis, Sewelson, and Immerman [HSI] claim the converse as well, but Allender [Al] points out that their proof is flawed and that the theorem cannot be proved by using techniques that relativize.

the last $n - k$ bits of x are zero. Then $C = \{x : A(x) = 1\}$ is a candidate selector for L, and $\mu_C(n) \le 2^{1+g^{-1}(n)} = O(\log n)$, so that L is uniformly log-sparse.

The fact that $L \in \mathrm{NP} - \mathrm{P}$ follows directly from the fact that $L' \in \mathrm{NEE} - \mathrm{EE}$. □

2.3. A language for which search does not reduce to decision. The following, which is the main lemma of this section, shows that the reduction of search to decision for a uniformly log-sparse language is possible only in the trivial case for which the language is already in P.

LEMMA 2.8. *Suppose L is a uniformly log-sparse language for which search reduces to decision. Then $L \in \mathrm{P}$.*

Proof. By assumption there exists an NP-relation ρ and a polynomial-time oracle machine W such that ρ defines L and W is a ρ-witness finder. We will construct a polynomial-time machine M that decides L. We begin by describing the idea informally.

The idea is to use W as a subroutine to find a witness. The difficulty is, of course, that W makes oracle queries about L itself. Not having access to an oracle for L, our machine M certainly cannot correctly answer these queries. To see how it can nonetheless exploit W, suppose for a moment that W is guaranteed to make only one oracle query in its entire computation on input x. Then M can try both possible answers. That is, it branches into a pair of parallel computations. In the first it answers the query by 0, in the second it answers it by 1, and in both it then runs W until W halts. Clearly, $x \in L$ if and only if at least one of these runs outputs a witness, and the strategy is polynomial time.

This idea extends to W making $O(\log n)$ queries. In reality, however, W could make polynomially many queries, so that this strategy is not efficient. This is the point at which we invoke the uniform log-sparseness of L (which we have not used so far). This implies that there are really only $O(\log n)$ effective queries that W can make; since W can write down only queries of polynomial length, we can use the log-sparse candidate selector of L to identify a set of at most $O(\log n)$ strings that include all the strings in L that W could possibly query, and we need branch only on these.

With this overall strategy in mind let us now specify the operation of M more precisely. Since ρ is an NP-relation, there exists a constant $c \in \mathrm{N}$ such that for all $x \in \{0, 1\}^*$ it is the case that $\rho(x) \subseteq \{0, 1\}^{|x|^c}$. We can assume that there is a constant $d > 0$ such that on any input $x \in \{0, 1\}^*$ the machine W will halt in $\le d|x|^d$ steps and output a string of length $|x|^c$, regardless of how the oracle queries of W are answered. We also assume (without loss of generality) that all queries made by W are distinct. Let C be a log-sparse candidate selector for L. Now, on input $x \in \{0, 1\}^n$ the machine M behaves as follows:

(1) M runs W on input x. Each time W makes an oracle query q, the machine M provides a response as follows:

(1.1) If $q \notin C$, then M responds with 0.

(1.2) Else it continues by trying in parallel both possible answers 0 and 1. That is, M branches into two parallel computations. In the first it lets the response to q be 0, and in the second it lets the response to q be 1. It then continues to run W in each computation.

In this manner M generates a number of parallel computations. After dn^d steps all of these computations have halted and each has yielded an n^c-bit output (the output of W).

(2) M now examines the set of outputs from the previous step. It accepts if at least one of these outputs w satisfies $\rho(x, w) = 1$ and rejects otherwise.

This completes the description of the machine M. The fact that it works should be clear, but for completeness let us spell it out.

First, to see that M accepts x if and only if $x \in L$, it suffices to check that on at least one of the parallel computations all oracle queries are correctly (that is, according to L) answered.

But Step (1.1) is obviously correct by definition of the candidate selector and in step (1.2) everything is being tried, so that one of the runs will certainly end up having all the right query answers.

The next thing to check is whether M runs in polynomial time. It suffices to show that the total number of parallel computations is $n^{O(1)}$. For this it suffices to show that the number of branches on any path is $O(\log n)$. We now argue the latter. First note that any query q has length at most dn^d (the running time of W). But branching occurs only when $q \in C$, and we have assumed that all W's queries are distinct. So the number of times branching occurs is at most the size of $\{q \in C : |q| \leq dn^d\}$, which is at most $\mu_C(dn^d) = O(\log n)$. This completes the proof. □

We can now put the pieces together to obtain the following result.

THEOREM 2.9. *If* EE $\neq$ NEE, *then there exists a language in* NP *for which search does not reduce to decision.*

Proof. By Lemma 2.7 there exists a uniformly log-sparse language $L \in$ NP $-$ P. By Lemma 2.8 search cannot reduce to decision for L. □

Note that the fact that search does not reduce to decision for L implies that L is not NP-complete. The existence of a non-NP-complete language in NP $-$ P can, however, be established by assuming only P $\neq$ NP (see [La]).

3. Deciders and their properties. Before extending the ideas of the previous section to interactive proofs and checking, we pause to develop some technical material. This material will be useful in proving the results of later sections. In particular, we introduce the notion of a decider, which will enable us to give a unified and more concise treatment of the rest of the results of this paper. We begin with the definition.

DEFINITION 3.1. Let D be a probabilistic, polynomial-time oracle machine. We say that D is a decider for language L if for each $x \in \{0, 1\}^*$ the following are true:

(1) If $x \in L$, then $D^L(x)$ accepts with probability $\leq 2/3$.

(2) If $x \notin L$, then the probability that $D^A(x)$ accepts is $\leq 1/3$ for all oracles A.

We note that L has a decider if and only if it is in function-restricted IP (see [BK]). So deciders are just a way of characterizing languages in frIP. They can also be viewed as checkers for YES instances. They are weaker that multiple-prover interactive proofs: the results of [FRS] imply that if L has a decider, then it has a multiple-prover interactive proof. For us the motivation of Definition 3.1 is to generalize the notion of a witness finder in the light of our proof of Lemma 2.8. The property of the witness finder that was important in that proof was that it was correct for $x \in L$ as long as oracle queries were answered correctly (i.e., according to L), and it was correct for $x \notin L$ no matter how oracle queries were answered. As for a witness finder, correctness of the decider on inputs $x \in L$ is guaranteed (except here only with high probability) as long as oracle queries are correctly answered. On the other hand, if $x \notin L$, then again correctness is guaranteed with high probability, *no matter how oracle queries are answered.* As we will see, these properties will suffice for us to approximately extend Lemma 2.8 to Lemma 4.3.

The error probability of 1/3 in Definition 3.1 is not always sufficient. It is convenient also to define the following.

DEFINITION 3.2. Let D be a probabilistic, polynomial-time oracle machine. We say that D is a *strong decider* for L if for each $x \in \{0, 1\}^*$ the following are true:

(1) If $x \in L$, then $D^L(x)$ accepts with probability $\geq (1 - 2^{-|x|})$.

(2) If $x \notin L$, then the probability that $D^A(x)$ accepts is $\leq 2^{-|x|}$ for all oracles A.

Standard error reduction, of course, says that strong deciders exist whenever deciders exist. For completeness let us state this as a proposition and provide a sketch of the proof.

PROPOSITION 3.3. *If L has a decider, then it has a strong decider.*

Proof. Let D be a decider for L. We define machine D' as follows. On input x, machine D' runs D on input x a total of $m = O(n)$ times, each time with independent coin tosses. The oracle queries made by D' are answered by D' by way of his own oracle (that is, if D makes oracle query q, then D' makes oracle query q and provides the answer received to D). D' outputs the majority vote of the outputs of D in the m trials. To see that D' is a strong decider for L, let $X_1^A, \dots, X_m^A$ denote the sequence of random variables representing the outcomes of D with oracle A in these successive trials. These are independent, and for each $i = 1, \dots, m$ they satisfy $\Pr[X_i^L = 1] \geq 2/3$ in the case that $x \in L$, and they satisfy $\Pr[X_i^A = 1] \leq 1/3$ for all oracles A in the case that $x \notin L$. An application of standard Chernoff bounds yields the desired conclusions. □

We saw in §2 that reducing search to decision for uniformly log-sparse languages is hard. Here we show that these same languages also do not have deciders unless they are in BPP.

LEMMA 3.4. *Suppose L is uniformly log-sparse and has a decider. Then* $L \in$ BPP.

Proof. By Proposition 3.3, L has a strong decider D. We show how to use D to construct a BPP machine M to decide L. The idea is very much the same as that in the proof of Lemma 2.8, with the decider here playing the role that the witness finder played in that proof. That is, on input x the machine M will run D on input x and will answer its oracle queries according to the same rules as those used in the proof of Lemma 2.8. M accepts if and only if the decider accepts on at least one of the parallel computations. The main difference (with respect to Lemma 2.8) lies in the fact that there is no way to tell whether a particular output of the decider is correct (in Lemma 2.8 one can always check whether or not the output of the witness finder is really a witness). Instead, the correctness of the procedure follows from the fact that the error probability of D is very small (2^{-n}). Details follow.

Let d be a constant such that D always halts in $\leq dn^d$ steps on inputs of length n. Let C be the log-sparse candidate selector for L. We assume without loss of generality that all oracle queries made by D are distinct. On input $x \in \{0, 1\}^n$ the machine M behaves as follows:

(1) M runs D on input x. Each time D makes an oracle query q, the machine M provides a response as follows:

(1.1) If $q \notin C$, then D responds with 0.

(1.2) Else it continues by trying in parallel both possible answers 0 and 1. That is, M branches into two parallel computations. In the first it lets the response to q be 0, and in the second it lets the response to q be 1. It then continues to run W in each computation.

In this manner M generates a number of parallel computations. After dn^d steps all of these computations have halted and each has yielded an output of 1 or 0 (the output of D).

(2) M now examines the set of outputs from the previous step. It accepts if at least one of these outputs is 1.

Since the answers in one of these parallel computations correspond to L, machine M accepts with probability $\geq (1 - 2^{-n})$ if $x \in L$. Now suppose $x \notin L$. Let $Q = \{q \in C : |q| \leq dn^d\}$, and let $\mathcal{A}$ denote the set of all subsets of Q. Each parallel computation of M corresponds to running D with some oracle $A \in \mathcal{A}$. It follows that the probability that M accepts is at most $\sum_{A \in \mathcal{A}} \Pr[D^A \text{ accepts}]$. By assumption D is a strong decider for L, so that we can bound this by $|\mathcal{A}| \cdot 2^{-n}$. But we claim that $|\mathcal{A}| \leq n^{O(1)}$, and so the probability that M accepts is $o(1)$, completing the proof. To justify the claim, note that $|\mathcal{A}| \leq 2^{|Q|}$ and $|Q| \leq \mu_C(dn^d) = O(\log n)$. □

Recall that BPEE denotes the class of languages accepted in time 2^{c2^n} for some constant $c \geq 0$ by a probabilistic machine with bounded error. By an argument analogous to that used in the proof of Lemma 2.7 we can show the following.

LEMMA 3.5. *If* NEE $\not\subseteq$ BPEE, *then there exists a uniformly log-sparse language in* NP − BPP.

Combining this with Lemma 3.4, we obtain the following theorem, which we will use in the next two sections.

THEOREM 3.6. *If* NEE $\not\subseteq$ BPEE, *then there exists a language in* NP *that does not have a decider.*

We remarked earlier that L has a decider if and only if $L \in$ frIP, and so Theorem 3.6 says that NEE $\not\subseteq$ BPEE implies NP $\not\subseteq$ frIP. In later sections we will use Theorem 3.6 to show that (NEE $\not\subseteq$ BPEE implies) NP $\not\subseteq$ compIP and check by showing that languages in compIP and Check have deciders (see Lemmas 4.3 and 5.2).

4. Competitive interactive proofs.

We begin by recalling the notion of an interactive proof. We then define competitive interactive proofs and present our results.

4.1. Interactive proofs.

Interactive proofs are extensions of NP ones, so let us begin by recalling the latter. An NP proof system for a language L is defined by a polynomial-time *verifier* V. We imagine this verifier talking to a prover. The parties receive a common input x, and the prover's goal is to convince the verifier to accept. To this end he is allowed to send the verifier a (single) message of length $n^{O(1)}$. The verifier's decision as to whether or not to accept is made as a function of the common input and this message (since the verifier is deterministic, this decision is a polynomial-time binary predicate ρ evaluated on the common input and the prover's message). In the case that $x \in L$ we ask that there exist some *deterministic* prover P who can convince the verifier to accept (this is the completeness condition). In the case that $x \notin L$ no prover should be able to convince the verifier to accept (this is the soundness condition). We usually specify an NP proof system as a pair (P, V), where P is a prover satisfying the completeness condition. Clearly, $L \in$ NP if and only if it possesses an NP proof system.

Interactive proofs, which were introduced by Goldwasser, Micali, and Rackoff [GMR], are a natural extension of such NP proof systems. Both parties are now allowed to be probabilistic. Moreover, they are allowed to interact (that is, they exchange messages for a polynomial number of rounds, and it is only at the end of this exchange that V decides whether or not to accept). We say that (P, V) is an interactive proof for a language L if (1) on common input a string in L, it is possible for P to induce V to accept with high probability, and (2) on common input a string not in L, there is no prover who can prevent V from rejecting with high probability.

Let us now proceed more formally. A party A in an interactive proof may be viewed as a (probabilistic) function of the common input and the conversion so far. The outcome of this function on input x (the common input) and c (transcript of conversation so far), which we denote by $A(x, c)$, is the next message computed by A (and sent to the other party).[3] We assume that the transcript of the conversation at any point may be uniquely parsed into its constituent messages. We may discuss the complexity of such parties in the usual way, viewing them as being computed by (probabilistic) TM's. For example, the verifier is a party computable by a probabilistic, polynomial-time TM. Complexity is measured as a function of the length of the common input (which we usually denote by n). The total number of moves (a move consists of a party computing and sending a message) as well as the length of all messages are assumed to be bounded by a polynomial in n. At the end of the interaction, the verifier accepts or rejects by applying a (deterministic) binary predicate to the common input and transcript of conversation. Suppose a prover A interacts with a verifier B on common

[3] When we say that this function is probabilistic, we mean that to any x, c party A actually associates a distribution on strings and $A(x, c)$ is a random element of this distribution.

input x. The "probability that B accepts in its interaction with A on common input x" is the probability the B accepts given the common input x and transcript $\alpha_1\beta_1 \cdots \alpha_{g-1}\beta_{g-1}\alpha_g$ chosen according to the following experiment: $\alpha_1 = A(x, \lambda)$, $\beta_1 = B(x, \alpha_1)$, $\alpha_2 = A(x, \alpha_1\beta_1)$, $\beta_2 = B(x, \alpha_1\beta_1\alpha_2), \ldots, \alpha_g = A(x, \alpha_1\beta_1 \cdots \alpha_{g-1}\beta_{g-1})$. (Here $g = g(n)$ is the total number of moves, λ is the empty string, and we are assuming for simplicity that A speaks first and last. The probabilities are over the random choices of both parties in this conversation.)

DEFINITION 4.1 (interactive proofs [GMR]). Let (P, V) be a pair of (probabilistic) functions. We say that (P, V) is an interactive proof system for language L if V is probabilistic, polynomial time and the following hold:

(1) For every $x \in L$ the probability that V accepts in its interaction with P on common input x is $\geq 2/3$.

(2) For every $x \notin L$ and every function $\hat{P}$ the probability that V accepts in its interaction with $\hat{P}$ on common input x is $\leq 1/3$.

The first condition is the completeness condition, and the second is the soundness condition.

We note the strength of the soundness condition: the quantification is over all functions $\hat{P}$ (we call them "cheating" or "dishonest" provers), even noncomputable ones.

We note that an NP proof system is a special kind of interactive proof system. Specifically, an NP proof system is an interactive proof system (P, V) in which both P and V are deterministic, the interaction is restricted to a single message from the prover to the verifier, and the probabilities in the completeness and soundness conditions are 1 and 0 (rather than $2/3$ and $1/3$), respectively. The addition of interaction and randomness, however, seems to add significantly to the language-recognition power of the system. It was established by Lund, Fortnow, Karloff, and Nisan [LFKN] that IP (the class of languages possessing interactive proofs of membership) contains the polynomial-time hierarchy, and Shamir [Sh] extended this to show that IP equals PSPACE.

4.2. Competitive interactive proofs. A basic complexity-theoretic question is to determine how efficient the prover P can be in an interactive proof (P, V) of a language L. Certainly, the prover would need at least the ability to decide the language himself. We define a competitive interactive proof system as one where the prover is allowed no more than this. Specifically, he must run in probabilistic polynomial time given access to L as an oracle. As we will see, competitive interactive proofs represent the natural generalization of the problem of decision versus search.

DEFINITION 4.2. Let P be a probabilistic polynomial-time oracle machine, and let V be a probabilistic polynomial-time machine. We say that (P, V) is a *competitive interactive proof system* for a language L if the following hold:

(1) For every $x \in L$ the probability that V accepts in its interaction with P^L on common input x is $\geq 2/3$.

(2) For every $x \notin L$ and every interactive TM $\hat{P}$ the probability that V accepts in its interaction with $\hat{P}$ on common input x is $\leq 1/3$.

The first condition is the completeness condition, and the second is the soundness condition. We call P a *competitive prover.*

We note that the soundness condition remains the same as in the definition of interactive proofs. In particular, we do not restrict the computational power of the cheating prover $\hat{P}$ in the case of $x \notin L$. Our goal is to understand the difficulty of providing a correct proof, and unrestricted soundness would appear to be an inherent property of proofs.

Competitive NP proof systems are defined in the natural way. That is, a *competitive* NP *proof system* is a competitive interactive proof system in which both parties are deterministic, the interaction is restricted to a single message from the prover to the verifier, and the probabilities in the completeness and soundness conditions are 1 and 0 (rather than 2/3 and 1/3), respectively. Equivalently, it is an NP proof system in which the prover is restricted to polynomial time plus an oracle for L. We now note that search reduces to decision for L if and only if L has a competitive NP proof system (the prover in the competitive NP proof system corresponds to the witness finder). It is in this sense that competitive interactive proofs are the natural extension of the problem of decision versus search.

4.3. An NP language not possessing a competitive interactive proof. In §2 we presented a language $L \in$ NP for which search is unlikely to reduce to decision. In other words, L is unlikely to have a competitive NP proof system. The truth, however, is that proving membership in NP languages remains hard even when interaction and randomness are allowed: we will show here that there is probably an NP language that does not even have a competitive interactive proof system. Given the results of §3 we need only the following lemma, which shows that any language possessing a competitive interactive proof also has a decider (or, equivalently, that compIP $\subseteq$ frIP).

LEMMA 4.3. *Suppose L has a competitive interactive proof system. Then it has a decider.*

Proof. Let (P, V) be a competitive interactive proof for L. We note that in probabilistic polynomial time we can run both P and V. So given an oracle A, the machine D on input x can sample the space of conversations between P^A and V on input x and accept if and only if the conversation obtained is accepting. Details follow.

Let $r(n)$ denote (a polynomial bound on) the number of coin tosses used by P and V on any input of length n. D picks, uniformly at random, an $r(n)$-bit string R_P and an $r(n)$-bit string R_V. D now runs P and V on common input x, using R_P as the coins for P and R_V as the coins for V. That is, assuming for example that P sends the first message, D would run P (with coins R_P) to get P's first message. D would then run V (with coins R_V) to get the response, and so on. Oracle queries made by P in this process are answered by D by way of his own oracle (that is, if P makes oracle query q, the D makes oracle query q and provides the answer he receives to P). Eventually D obtains the output of V (1 if V accepts and 0 otherwise) and outputs this value. Given any particular oracle A, it is the case that $D^A(x)$ is a 0/1 random variable, and clearly the probability that it is 1 equals the probability that V accepts in its interaction with P^A on common input x. By the assumption that (P, V) was a competitive interactive proof for L it follows that $\Pr[D^L(x) = 1] \geq 2/3$ in the case that $x \in L$ and that $\Pr[D^A(x) = 1] \leq 1/3$ for all oracles A in the case that $x \notin L$. □

Combining Lemma 4.3 and Theorem 3.6 yields the theorem.

THEOREM 4.4. *If* NEE $\not\subseteq$ BPEE, *then there exists a language in* NP *that does not have a competitive interactive proof.*

We note that we have done more than simply show that interactive proofs may be more powerful than competitive ones, because the language L of Theorem 4.4 is in NP, a subclass of IP that possesses particularly simple interactive proofs. To show only that interactive proofs are more powerful than competitive ones, it would suffice to present a language in IP (but not necessarily in NP) that does not have a competitive interactive proof. This can be done under weaker assumptions, by an extension of the same argument we used previously. For example, let

$$\text{EESPACE} = \bigcup_{c \geq 0} \text{SPACE}(2^{c2^n}).$$

Then we can show that if EESPACE $\not\subseteq$ BPEE, then there exists a language in IP that does not possess a competitive interactive proof.

In general, to construct a language L that lies in some particular complexity class C but does not possess a competitive interactive proof, it suffices to assume that the "double-exponential counterpart" of C is not contained in BPEE. We put the phrase "double-exponential counterpart" in quotation marks because it, of course, does not always make sense (many classes have no such counterpart). But there are many natural classes (such as those used here) for which this paradigm does make sense.

4.4. Zero-knowledge aspects. The competitive aspects of zero-knowledge proofs may also be worth investigating. To initiate such an investigation, let us try to discuss briefly what one can easily infer from known work and what are the open questions.

Do NP languages have competitive zero-knowledge interactive proofs? In general, of course, they probably do not since by Theorem 4.4 they probably do not even have competitive interactive proofs (let alone ZK ones). An appropriate question then is whether NP languages that possess competitive interactive proofs also possess competitive zero-knowledge interactive proofs. The answer depends on the kind of zero-knowledge one considers and on the kind of cryptographic assumptions one is willing to make.

Let us first consider computational ZK. The result of Goldreich, Micali, and Wigderson [GMW] implies that NP-complete languages have competitive ZK interactive proofs, given the existence of one-way functions (more generally, it implies that if search reduces to decision for L, then L has a competitive ZK interactive proof, given the existence of one-way functions). We do not know whether the assumption that there exist one-way functions suffices to show that *any* language that possesses a competitive interactive proof also possesses a competitive ZK interactive proof. But we do know that the latter conclusion may be established with stronger assumptions such as the existence of ideal secure circuit evaluation or the existence of oblivious transfer. This follows from the result of Kilian [Ki] (and we refer the reader to that paper for details on what exactly are these assumptions).

All statistical ZK languages known to possess competitive interactive proofs are also known to possess statistical ZK competitive interactive proofs (these languages are graph isomorphism [GMW], graph nonisomorphism [GMW], and quadratic nonresiduosity [GMR]). We do not, of course, know whether or not quadratic residuosity has a competitive statistical ZK interactive proof, given that we do not know whether or not it has a competitive interactive proof at all.

5. Program checking. Blum and Kannan [BK] introduce the notion of program checkers. Informally, a checker for a function f is a probabilistic, polynomial-time oracle machine that receives as an oracle a program P that purports to compute f. The checker also receives an input x. If the program is entirely correct (that is, $P(y) = f(y)$ for all y), then the checker is supposed to accept with high probability. However, if the program disagrees with f on the particular input x provided to the checker, then the checker should reject with high probability. The definition follows. We note that by a "program" we mean a deterministic machine that halts on all inputs. We also recall that the characteristic function of a language L is the function $\chi_L : \{0, 1\}^* \to \{0, 1\}$ defined by $\chi_L(x) = 1$ if $x \in L$ and 0 otherwise.

DEFINITION 5.1 [BK]. Let C be a probabilistic polynomial-time oracle TM. C is a *checker* for $f : \{0, 1\}^* \to \{0, 1\}$ if for all programs P and all $x \in \{0, 1\}^*$ the following hold:

(1) If $P(y) = f(y)$ for all $y \in \{0, 1\}^*$, then $C^P(x)$ accepts with probability $\geq 2/3$.

(2) If $P(x) \neq f(x)$, then the probability that $C^P(x)$ accepts is $\leq 1/3$.

We say C is a checker for a language L if it is a checker for the characteristic function of L. L is *checkable* if it has a checker.

The definition is close in spirit to that of (competitive) interactive proofs, but there are two important differences. First, unlike interactive proofs, checking is a symmetric notion in which the checker for language L must be able to determine that P is correct on x not only when $x \in L$ but also when $x \notin L$. Second, programs are history-independent objects, whereas (cheating) provers are not. Thus if L and $\bar{L}$ both have competitive interactive proofs, then L has a checker, whereas we do not know whether or not every checkable language has a competitive interactive proof.

Checkers are also related to multiple-prover interactive proofs [BGKW]. In particular, results of [FRS] imply that the class of languages that possess checkers is contained in MIP $\cap$ coMIP (where MIP is the class of languages possessing multiple-prover interactive proofs of membership). We note that MIP = NEXP by the result of [BFL].

Blum and Kannan [BK] showed that Check = frIP $\cap$ cofrIP. It follows that checkable languages have deciders. For completeness, however, let us see this directly.

LEMMA 5.2. *Suppose L has a checker. Then it has a decider.*

Proof. Let C be a checker for L. Let D be the probabilistic polynomial-time oracle machine that on input x works as follows. D begins by querying its oracle with the string x. If the oracle returns 0, then D rejects. Else it runs C on input x, uses its own oracle (denoted A) to answer C's oracle queries, and accepts if and only if C accepts. We claim that D is a decider for L. To see this we need to check that for each $x \in \{0, 1\}^*$ the two conditions of Definition 3.1 hold. The first condition is clear. To see that the second is true, suppose $x \notin L$, and suppose first that $A(x) = 0$. In this case D rejects with probability 1. Now suppose $A(x) = 1$. Then the probability that D accepts is at most 1/3 because C is a checker. □

Combining Lemma 5.2 and Theorem 3.6 yields the theorem.

THEOREM 5.3. *If* NEE $\not\subseteq$ BPEE, *then there exists a language in* NP *that is not checkable.*

Similarly, if EESPACE $\not\subseteq$ BPEE, then there exists a language in PSPACE that is not checkable.

We recall that a language L is *coherent* if the membership of x in L can be decided in probabilistic polynomial time and bounded error by a machine (called the *examiner*) that has access to L as an oracle but is allowed to query this oracle only on points different from x. If L is not coherent, we say it is incoherent. Previous negative results on checking were established by first exhibiting incoherent sets and then exploiting Yao's observation that any incoherent set is uncheckable (see [Ya], [BF]). We note that our (stronger) results are obtained more directly. Moreover, our techniques indicate that even within NP the class of coherent sets could be much larger than the class of checkable ones. Let us sketch why this is so.

The disjoint union of languages A and B, denoted $A \oplus B$ is $\{0x : x \in A\} \cup \{1x : x \in B\}$; this construct is widely used in complexity theory (see, e.g., [BD], [HH]). It is easy to see (see [BF]) that $L \oplus L$ is coherent for any language L. It is also easy to see that the transformation $L \mapsto L \oplus L$ preserves many complexity characteristics of L, for example, membership in NP, compNP, compIP, Check, frIP. In particular, combining this observation with Theorem 5.3 yields the claimed separation.

THEOREM 5.4. *If* NEE $\not\subseteq$ BPEE, *then there exists a language in* NP *that is coherent but not checkable.*

6. Towards competitive proofs for quadratic residuosity. In this section we return to the unresolved question of whether the language of quadratic residuosity has a competitive interactive proof system, and we present a special case of the problem for which competitive proofs are possible.

6.1. Definitions. We will be looking at *promise* problems rather than problems of language membership. The difference is that in the former we begin with a promise that the input already belongs to some set, and we have only to "decide" whether or not it falls in a given subset of this set. Such problems have been considered in many works; see, e.g., [ESY]. The formalization we use is different from (but equivalent to) the ones used in these works and is as follows. The problem is specified by a pair of disjoint sets (A, B). Intuitively, the input is promised to be in $A \cup B$, and we have to decide whether it is in A or in B. Corresponding to promise problems are promise oracles that are guaranteed to be correct only when the promise is true.

DEFINITION 6.1. *A promise problem* is a pair of disjoint sets (A, B). *A promise oracle* (for a promise problem (A, B) is an oracle that given a query q returns 1 if $q \in A$ and 0 if $q \in B$.

Note that whereas promise problems are, intuitively, easier than language-recognition problems, promise oracles are correspondingly weaker than (normal) oracles. In particular, a promise oracle for (A, B) is weaker than an oracle for just A (or B) in that its response on queries outside $A \cup B$ is indeterminate.

A competitive interactive proof for a promise problem (A, B) is just an interactive proof that $x \in A$ given that $x \in A \cup B$ and having the property that the competitive prover gets only a promise oracle for (A, B). The more formal definition follows.

DEFINITION 6.2. Let P be a probabilistic polynomial-time oracle machine, and let V be a probabilistic polynomial-time machine. We say that (P, V) is a competitive interactive proof for promise problem (A, B) if the following hold:

(1) For every $x \in A$ and every promise oracle O for (A, B), the probability that V accepts in its interaction with P^O on common input x is $\geq 2/3$.

(2) For every $x \in B$ and every interactive TM $\hat{P}$, the probability that V accepts in its interaction with $\hat{P}$ on common input x is $\leq 1/3$.

6.2. Results. We recall that $x \in Z_N^*$ is a *quadratic residue* (or *square*) mod N if $x \equiv y^2$ (mod N) for some $y \in Z_N^*$ and a *quadratic nonresidue* (or *nonsquare*) mod N otherwise. Also, recall from §1.2 that

$$\mathrm{QR} = \{(x, N) : x \text{ is a square mod } N\},$$

$$\mathrm{QNR} = \{(x, N) : x \text{ is a nonsquare mod } N\}.$$

The special case we are interested in is when N is the product of a constant number of distinct odd primes. To be more precise, first define

$$\mathrm{QR}_s = \{(x, N) \in \mathrm{QR} : N \text{ is a product of } s \text{ distinct odd primes}\},$$

$$\mathrm{QNR}_s = \{(x, N) \in \mathrm{QNR} : N \text{ is a product of } s \text{ distinct odd primes}\}.$$

We will present a competitive interactive proof that $(x, N) \in \mathrm{QR}_s$, given that it is already in $\mathrm{QR}_s \cup \mathrm{QNR}_s$. Note that QR_s and QNR_s are not complements of each other, so that, formally, we are talking of a competitive interactive proof for the promise problem $(\mathrm{QR}_s, \mathrm{QNR}_s)$ in the sense of Definition 6.2.

THEOREM 6.3. *Let s be an integer* ≥ 1. *Then the promise problem* $(\mathrm{QR}_s, \mathrm{QNR}_s)$ *possesses a competitive interactive proof.*

In related work, Kompella and Adleman [KA] present checkers for this same special case of quadratic residuosity when the modulus is the product of a constant number of primes (i.e., they present checkers for the promise problem $(\mathrm{QR}_s, \mathrm{QNR}_s)$). Their construction does not, however, extend to competitive interactive proofs, because the correctness of their checker uses the fact that a program (in contrast to a cheating prover) is history independent.

To prove Theorem 6.3 we begin by recalling some basic number-theoretic facts. We refer the reader to [An], [NZ] for number-theoretic background and justification of these facts.

For $x \in Z_N^*$ we let $Q_N(x) = 0$ if x is a quadratic residue mod N, and we let $Q_N(x) = 1$ otherwise. Suppose $N = p_1 \cdots p_s$, where $p_1, \ldots, p_s$ are distinct odd primes. Define the binary relation ($\simeq$) on Z_N^* by

$$x \simeq y \quad \text{iff } \forall i : 1 \leq i \leq s : Q_{p_i}(x \bmod p_i) = Q_{p_i}(y \bmod p_i).$$

This is an equivalence relation. The equivalence class of x under this relation (namely, $\{y \in Z_N^* : x \simeq y\}$) is called its *residue class.* The product xy mod N of two elements $x, y \in Z_N^*$ is a square mod N if x, y are from the same residue class and is a nonsquare mod N otherwise. The total number of residue classes is 2^s, and they are all of the same size. We will denote them by $R_N^1, \ldots, R_N^{2^s}$, with the convention that the last $R_N^{2^s}$, is the class of quadratic residues mod N.

We recall that there exists a competitive interactive proof for quadratic nonresiduosity. We will exploit this fact by reducing the proof of $(x, N) \in \mathrm{QR}_s$ to a polynomial number of proofs of nonresiduosity in such a way that the prover need use only probabilistic, polynomial time and a promise oracle for $(\mathrm{QR}_s, \mathrm{QNR}_s)$. The first step is the following definition.

DEFINITION 6.4. Let N be a product of s distinct odd primes, and let $t = 2^s$. We call a vector $(y_1, \ldots, y_{t-1}) \in (Z_N^*)^{t-1}$ *representative* of Z_N^* if the following conditions hold:

(1) y_i is a nonsquare mod N for each $i = 1, \ldots, t-1$.

(2) $y_i y_j$ mod N is a nonsquare mod N for each pair of indices i, j satisfying $1 \leq i < j \leq t-1$.

This leads us to a way to reduce a residuosity test to a collection of nonresiduosity tests as long as we are in possession of a representative vector.

PROPOSITION 6.5. *Let N be a product of s distinct odd primes, and let $x \in Z_N^*$. Suppose $\vec{y} = (y_1, \ldots, y_{t-1})$ is representative of Z_N^* (where $t = 2^s$). Then $(x, N) \in \mathrm{QR}_s$ if and only if xy_i* mod *N is a nonsquare mod N for each $i = 1, \ldots, t-1$.*

Proof. xy_i mod N will be a nonsquare mod N for each $i = 1, \ldots, t-1$ if and only if its residue class differs from the residue class of y_i for each $i = 1, \ldots, t-1$. But since $\vec{y}$ is representative, this happens if and only if x is a square mod N. □

To use this, however, we have to be able to get representative vectors. It suffices to show how the prover can construct a representative vector and then convince the verifier that it is indeed representative, all by using only probabilistic, polynomial time and a promise oracle for $(\mathrm{QR}_s, \mathrm{QNR}_s)$.

PROPOSITION 6.6. *There is a probabilistic, polynomial-time oracle machine R that on input $(x, N) \in \mathrm{QR}_s \cup \mathrm{QNR}_s$ and access to a promise oracle for $(\mathrm{QR}_s, \mathrm{QNR}_s)$ outputs either a representative vector for Z_N^* or the special symbol $\perp$, with the probability of the latter event being at most* $1/4$.

Proof. R picks at random $\vec{y}_1, \ldots, \vec{y}_m \in (Z_N^*)^{t-1}$, where $t = 2^s$ and m is a constant to be defined later. For each $i = 1, \ldots, m$ the machine R then uses the promise oracle to test whether or not the conditions of Definition 6.4 hold for $\vec{y}_i$. If some vector $\vec{y}_i$ passes the test, then the first such vector is output. If all vectors fail the test, then R outputs $\perp$. The probability that a particular vector passes the test is $(t-1)!/t^{t-1}$, which is a positive constant. So it suffices to choose m to be a constant such that

$$\left[1 - \frac{(t-1)!}{t^{t-1}}\right]^m \leq \frac{1}{4}.$$

(A (crude) calculation shows that $m = O((2e)^t)$ suffices.) That R runs in probabilistic, polynomial time is clear. □

We can now proceed to describe the protocols. We begin by recalling (following [GMR]) the basic (competitive) protocol to prove nonresiduosity.

PROTOCOL QNR.

Input: (x, N) and 1^k

(V1) V picks at random $c_1, \dots, c_k \in \{0, 1\}$ and $r_1, \dots, r_k \in Z_N^*$, sets $z_i = x^{c_i} r_i^2 \bmod N$ (for $i = 1, \dots, k$), and sends $z_1, \dots, z_k$ to P.

(P1) P sets d_i to 0 if z_i is a quadratic residue mod N and to 1 otherwise (for $i = 1, \dots, k$) and sends $d_1, \dots, d_k$ to V.

(V2) V accepts if and only if $c_i = d_i$ for all $i = 1, \dots, k$.

PROPOSITION 6.7. *Protocol* QNR *has the following properties*:

(1) *If* $(X, N) \in$ QNR, *then the probability that* V *accepts in its interaction with* P *is* 1.

(2) *If* $(x, N) \notin$ QNR, *then for any* $\hat{P}$ *the probability that* V *accepts in its interaction with* $\hat{P}$ *is* $\leq 2^{-k}$.

(3) P *is competitive* (*that is, it runs in probabilistic, polynomial time, given an oracle for* QNR).

Proof. The first two items follow from basic properties of modular residues, and we refer the reader to [GMR] for proofs. The last item is clear. □

We now proceed to the competitive interactive proof for QR_s. We will use Protocol QNR as a subprotocol.

PROTOCOL QR(s)

Input: $(x, N) \in \mathrm{QR}_s \cup \mathrm{QNR}_s$

Notation: We let $t = 2^s$.

(P1) P runs the algorithm of Proposition 6.6 and sends the output to V.

(V1) If V receives $\perp$ from P, then it rejects. If instead it receives a vector $\vec{y} = (y_1, \dots, y_{t-1}) \in (Z_N^*)^{t-1}$, then the parties proceed to the next step.
Subprotocol: P uses Protocol QNR (with security parameter k set to 2) to prove the following to V:
(1) y_i is a nonsquare mod N for each $i = 1, \dots, t-1$.
(2) $y_i y_j$ mod N is a nonsquare mod N for each pair of indices i, j satisfying $1 \leq i < j \leq t-1$.
(3) $x y_i$ mod N is a nonsquare mod N for each $i = 1, \dots, t-1$.
(This is a total of $(t-1)(t+2)/2$ invocations of the QNR protocol.)

(V2) V accepts if and only if each of the preceding subproofs was accepting.

The correctness of the protocol follows from the results established in the preceding. Details follow.

Suppose $(x, N) \in \mathrm{QR}_s$. Proposition 6.6 implies that the parties get a representative vector and proceed to the subprotocol with probability $\geq 3/4$. Definition 6.4 and Proposition 6.5 imply that the inputs to the nonresiduosity subproofs are all indeed nonsquares mod N, and thus Proposition 6.7 implies that these subproofs all succeed with probability 1. So V accepts with probability $\geq 3/4$.

Suppose $(x, N) \in \mathrm{QNR}_s$. If $\hat{P}$ sends $\perp$ in its first step, then V rejects; so suppose $\hat{P}$ sends a vector $\vec{y} = (y_1, \dots, y_{t-1}) \in (Z_N^*)^{t-1}$. If $\vec{y}$ is not representative, then by Definition 6.4 either there is an i such that y_i is a square mod N or there is a pair $i < j$ such that $y_i y_j$ mod N is a square mod N. In either case the corresponding nonresiduosity subproof fails with probability $\geq 3/4$, and V rejects. So suppose $\vec{y}$ is representative. But the Proposition 6.5 implies there is an i such that $x y_i$ mod N is a square mod N. So the corresponding nonresiduosity subproof fails with probability $\geq 3/4$, and V again rejects.

The competitiveness of P follows from Propositions 6.6 and 6.7.

The reason this does not extend to arbitrary N is, of course, that the number of residue classes is in general exponential in the length of N, and in polynomial time we could not even write down a representative list. On the other hand, working through the proofs shows that the result does extend to the case for which $s : \mathbf{N} \rightarrow \mathbf{N}$ is a polynomial-time-computable function of N that is bounded above by $\lg \lg \lg N = \lg \lg |N|$. For simplicity we have stuck to the case of constant s.

Clearly, the weakness of this result is in the promise that N is already a product of (exactly) s odd primes; this is what may be hard to prove competitively if one wants a competitive interactive proof of QR.

7. Open questions. *Quadratic residuosity.* We think that the most interesting open question is whether or not the language of quadratic residuosity has a competitive interactive proof. Conditional results on the subject would also be interesting; for example, could one show that if quadratic residuosity has a competitive interactive proof, then factoring is reducible (in probabilistic, polynomial time) to deciding quadratic residuosity? (Note that an affirmative answer to this last question would imply that if QR has a competitive interactive proof, then it has an NP proof through the simple factorization witness.)

Reducing assumptions. Another open question is whether one can reduce the assumptions required for our results. In particular, can one show that there is a language for which search does not reduce to decision given P $\neq$ NP or even E $\neq$ NE? Or could cryptographic assumptions such as the existence of one-way functions be used to establish the existence of languages in IP that do not have competitive interactive proofs?

Other settings. What is the relationship of decision to search in the context of optimization problems and approximation algorithms, and does search reduce to decision in this setting? For example consider the traveling salesman problem. Let $\omega(G)$ denote the weight of an optimal tour on a (weighted) graph G, and suppose $\mu \geq 1$ is a constant. Suppose A is an oracle satisfying $\omega(G) \leq A(G) \leq \mu \cdot \omega(G)$ for all graphs G. Is there a polynomial-time procedure that with oracle access to A and input G, outputs a tour (in G) of weight at most $\mu \cdot \omega(G)$?

Perfect completeness. An interactive proof (P, V) for L is said to have perfect completeness if the probability of acceptance in the completeness condition is 1. We know that any language L possessing an interactive proof also possesses one with perfect completeness [FGMSZ]. Does any language possessing a competitive interactive proof also possess a competitive interactive proof with perfect completeness? (One of the motivations for this question is the fact that our competitive proof for the special case of quadratic residuosity in §6 does not possess perfect completeness.)

Zero knowledge. We discussed the open questions in §4.4.

Acknowledgments. Many people pointed out to us the error in [HSI]: we thank Eric Allender, Juris Hartmanis, Jack Lutz, and Osamu Watanabe in this regard. We thank Muli Safra, Lance Fortnow, and Jack Lutz for helpful discussions, and we thank Silvio Micali for suggesting the term "competitive interactive proofs." We thank Oded Goldreich for many helpful comments on this paper. We thank Satish Thate for drawing our attention to the results of [La] mentioned in §2. Finally, we thank a pair of (anonymous) referees for many valuable comments on the paper.

REFERENCES

[An] D. ANGLUIN, *Lecture Notes on the Complexity of Some Problems in Number Theory*, Tech. Report 243, Department of Computer Science, Yale University, New Haven, CT, August 1982.

[Al] E. ALLENDER, *Limitations of the Upward Separation Technique*, Math. Systems Theory, 24 (1991), pp. 53–67.

[BFL] L. Babai, L. Fortnow, and C. Lund, *Non-deterministic exponential time has two-prover interactive protocols*, Computational Complexity, 1 (1991), pp. 3–40.

[BBFG] R. Beigel, M. Bellare, J. Feigenbaum, and S. Goldwasser, *Languages that are easier than their proofs*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1991.

[BF] R. Beigel and J. Feigenbaum, *Improved Bounds on Coherence and Checkability*, Tech. Report TR-819, Department of Computer Science, Yale University, New Haven, CT, September 11, 1990.

[BG] M. Bellare and S. Goldwasser, *The Complexity of Decision versus Search*, Tech. Memo TM-444, Laboratories of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, April 1991.

[BP] M. Bellare and E. Petrank, *Making zero-knowledge provers efficient*, in Proc. 24th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1992.

[BCGL] S. Ben-David, B. Chor, O. Goldreich, and M. Luby, *On the theory of average case complexity*, J. Comput. System Sci., 44 (1992), pp. 193–219.

[BGKW] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson, *Multiprover interactive proof systems: How to remove intractability assumptions*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988.

[BK] M. Blum and S. Kannan, *Designing programs that check their work*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989.

[BD] A. Borodin and A. Demers, *Some Comments on Functional Self-Reducibility and the* NP *Hierarchy*, Tech. Report TR76-284, Department of Computer Science, Cornell University, Ithaca, NY, 1976.

[Co] S. Cook, *The complexity of theorem proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971.

[ESY] S. Even, A. Selman, and Y. Yacobi, *The complexity of promise problems with applications to public-key cryptography*, Inform. and Control, 2 (1984), pp. 159–173.

[FRS] L. Fortnow, J. Rompel, and M. Sipser, *On the power of multiprover interactive protocols*, in Proc. 3rd IEEE Conference on Structures, IEEE Computer Society, Washington, DC, 1988.

[FGMSZ] M. Furer, O. Goldreich, Y. Mansour, M. Sipser, and S. Zachos, *On completeness and soundness in interactive proof systems*, in Advances in Computer Research Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT.

[GMW] O. Goldreich, S. Micali, and A. Wigderson, *Proofs that yield nothing but their validity*, J. Assoc. Comput. Mach., 1991.

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

[HH] J. Hartmanis and L. Hemachandra, *Complexity classes without machines*, Theoret. Comput. Sci., 58 (1988), pp. 129–142.

[HSI] J. Hartmanis, V. Sewelson, and N. Immerman, *Sparse sets in* NP-P: EXPTIME *versus* NEXPTIME, Inform. and Control, 65 (1985), pp. 158–181.

[IN] R. Impagliazzo and M. Naor, *Decision trees and downward closures*, in Proc. 3rd IEEE Conference on Structures, IEEE Computer Society, Washington, DC, 1988.

[IS] R. Impagliazzo and M. Sudan, Private Communication, May 1991.

[IT] R. Impagliazzo and G. Tardos, *Decision versus search problems in super-polynomial time*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989.

[KUW] R. Karp, E. Upfal, and A. Wigderson, *The complexity of parallel search*, J. Comput. System Sci., 36 (1988), pp. 225–253.

[Ki] J. Kilian, *Achieving zero-knowledge robustly*, in Advances in Cryptology—Proc. CRYPTO '90, A. J. Menezes and S. Vanstone, eds., Lecture Notes in Computer Science 537, Springer-Verlag, Berlin, 1990.

[KA] K. Kompella and L. Adleman, *Fast checkers for cryptography*, in Advances in Cryptology—Proc. CRYPTO '90, A. J. Menezes and S. Vanstone, eds., Lecture Notes in Computer Science 537, Springer-Verlag, Berlin, 1990.

[Kr] H. Krawczyk, Personal Communication.

[La] R. Ladner, *On the structure of polynomial time reducibility*, J. Assoc. Comput. Mach., 22 (1975), pp. 155–171.

[LFKN] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1990.

[NZ] I. Niven and H. Zuckerman, *An Introduction to the Theory of Numbers*, John Wiley, New York, 1960.

[Se] A. SELMAN, *Natural self-reducible sets*, SIAM J. Comput., 17 (1988), pp. 989–996.

[Sh] A. SHAMIR, IP=PSPACE, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1990.

[Sp] D. SPIELMAN, Private Communication via Beigel and Feigenbaum, June 1991.

[Va] L. VALIANT, *On the Relative Complexity of Checking and Evaluating*, Tech. Report LS29JT, University of Leeds, Leeds, England, October 1974.

[Ya] A. YAO, *Coherent functions and program checkers*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1990.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 120–132, February 1994

009

HOMEOMORPHISM OF 2-COMPLEXES IS GRAPH ISOMORPHISM COMPLETE *

JOHN SHAWE-TAYLOR[†] AND TOMAŽ PISANSKI[‡]

Dedicated to A.W. Tucker

Abstract. It is shown that the problem of determining whether two 2-complexes are homeomorphic is isomorphism-complete.

Key words. 2-complex, homeomorphism, graph, isomorphism, computational complexity, simplicial complex, triangulation

AMS subject classifications. 57M20, 57Q15, 68 15, 68Q25, 68R10

1. Introduction. There has been much interest focused on the position of GRAPH ISOMORPHISM in the complexity hierarchy. As yet it has neither been shown to be in P or to be NP-complete. As such it is one of the few "classical" problems that have not been classified as either "easy" or "hard". Various adaptations of the problem have been classified as NP-complete, while for others polynomial algorithms have been found. For example, Luks [4] has shown that if the valency is bounded by a constant then the problem is in P, while Lubiw [3] and others have considered slight adaptations that result in the problem becoming NP-complete. Thus subgraph isomorphism is NP-complete, as is the problem of determining the existence of an isomorphism with restrictions and the existence of a fixed point free automorphism.

The above considerations have led to the introduction of the class ISOMORPHISM-complete, which has been studied by Booth and Colbourn [1] and Lubiw [3]. They have identified a number of problems in this class—among others, automorphism with a fixed number of restrictions, finding generators of Aut(G), calculating $|\mathrm{Aut}(G)|$, Automorphism with 1 edge fixed, and so on. This paper extends the class of ISOMORPHISM-complete to include a new problem with a novel flavor—namely, 2-complex homeomorphism or homeomorphism between the carriers of 2-complexes. This problem has not been previously studied in the context of complexity theory, though it was shown to be decidable by Whittlesey [9], who worked under A. W. Tucker on this problem.

The paper is organized as follows. In §2 we introduce the definitions of the various structures and classes and state the main theorem and some known results that we will require. Section 3 develops the techniques that we require to complete our algorithm and proof. This is then presented in §4. We finish with some conclusions and directions for future research.

2. Definitions. In order to reduce a 2-complex homeomorphism to an isomorphism oracle, we will build a canonical invariant for a 2-complex. The invariant structure will be a labeled graph, which will encapsulate sufficient information for the construction of a topological space homeomorphic to the original complex. This will enable the definition of an algorithm to determine whether two complexes are homeomorphic. The algorithm first constructs the invariant graph for both complexes and then tests, using the isomorphism oracle, whether the two graphs obtained are isomorphic.

Note that a related problem of determining whether two complexes have isomorphic fundamental groups is algorithmically unsolvable. It reduces to the well-known word problem for group presentations.

*Received by the editors May 6, 1991; accepted for publication (in revised form) August 21, 1992. Supported in part by the Research Council of Slovenia Contract DMS-8717441.

[†] Department of Computer Science, Royal Holloway and New Bedford College, University of London, Egham, Surrey TW20 0EX, United Kingdom.

[‡] Department of Mathematics, University of Ljubljana, Jadranska 19, 61111 Ljubljana, Slovenia.

DEFINITION 1. A set X together with a collection $\mathcal{O}$ of subsets of X is a *topological space* if $\mathcal{O}$ contains $\emptyset$ and X and is closed under finite intersections and arbitrary unions of sets. The collection $\mathcal{O}$ are termed *open* sets.

A mapping h between two topological spaces X and Y is a *homeomorphism* if it is a bijection on both elements and open sets.

It should be noted that the definition of topology and, in particular, the Euclidean topologies considered in simplicial complexes are not finite or even countably infinite and so are not obviously open to combinatorial manipulation. This is even more apparent in the case of homeomorphisms of 2-complex carriers, which are not a priori constrained by the finite descriptions that we give of a complex. The results obtained here show that not only can we find a finite combinatorial way of describing the topologies considered, but that it enables us to assess the computational complexity of the homeomorphism problem for 2-complexes. In this context it is worth mentioning that changes in the precise combinatorial description used would affect the details of the algorithm and construction, but the significance of the results is not in the detailed choice of representation, but in the fact that the topological properties can be captured at all and, in particular, in such a way that the computational complexity can be determined.

We begin with the definition of a (purely) 2-dimensional complex. We will follow the notation and definitions of Gross and Tucker [2].

DEFINITION 2. A k-simplex is the convex hull of $k + 1$ affinely independent points in Euclidean n-space.

A *geometric simplicial complex* K is a finite collection of simplices in $\mathcal{R}^n$ satisfying two conditions:

1. Every face of every simplex in K is a simplex of K.
2. The intersection of any two simplices in K is a simplex in K.

The *carrier* of a simplicial complex K is the topological space with point set $|K| = \bigcup_{S \in K} S$.

An *abstract simplicial complex* $K = (V, C)$ is a finite set V together with a collection C of subsets of V whose union is V and which is closed under taking subsets.

The *dimension* of a complex is the dimension of the largest simplex or the size of the largest subset in the collection C minus one.

A *purely k-complex* is a simplicial complex for which each simplex is contained in a k-dimensional simplex (each subset is contained in a subset of size $k + 1$).

A *triangulation* of a topological space X is a homeomorphism h from the carrier of a simplicial complex K to the space X.

Every geometric simplicial complex gives rise to an abstract simplicial complex by taking the sets to be the vertices of simplices. Likewise for every abstract simplicial complex a geometric complex can be constructed so that the simplices correspond to the subsets of the abstract complex. The corresponding dimensions of the abstract and geometrical complexes agree, and one is a pure k-complex if and only if the other is. Further, if two geometrical complexes have the same abstract complex, they are homeomorphic. In view of the close relation between an abstract and geometrical complex, we will use the term complex ambiguously to refer to either. In fact, we extend this ambiguity to the carriers of complexes and hence to spaces that have a triangulation.

In this paper we are interested in (≤ 2)-dimensional complexes. We now state our main result.

THEOREM 2.1. *Homeomorphism of* 2*-complexes is graph isomorphism complete.*

Proof. We must show that homeomorphism of 2-complexes is equivalent to graph isomorphism modulo polynomial time reductions.

Given an instance of graph isomorphism, we can view the graph as a 2-complex by placing a distinct label on all the 2-valent vertices. The complexes constructed will be homeomorphic if and only if the the original graphs were isomorphic.

The reverse direction of the proof will require significantly more effort. □

To motivate our approach we first consider what is known about simpler complexes and their carriers.

Example 1. 1. *Closed surfaces* can be constructed from polygons so that all edges involved are identified in pairs and the resulting space is connected. Each surface can be characterized by a single integer (or a pair of an integer (*genus*) and a bit (*orientability type*)). Hence homeomorphism involves computing the genus and orientability. This can be done using Euler's formula for the genus and by attempting to fix an orientation.

2. *Disconnected closed surfaces.* Finite number of surfaces. Homeomorphism can be decided as above followed by a comparison of two sets of numbers.

3. *Surfaces with boundary.* They can be characterized by a triple: *genus, orientability type, number of boundary components.* Homeomorphism involves computing the genus, orientability, and number of boundary components. This can again be done using a straightforward algorithm. This characterization can be extended to the disconnected case as above.

We now introduce some more definitions that will prove useful in our reduction.

DEFINITION 3. Each graph G and a natural number d give rise to a d-complex $K(G, d)$ so that each k-clique $(0 < k \leq d + 1)$ determines a $(k - 1)$-simplex, and the incidence relation is defined in the obvious way.

For a vertex v of a complex K we define

- Star(v) to be the minimal subcomplex of K that contains every simplex incident with v;
- Link(v) to be the maximal subcomplex of Star(v) that does not contain v itself.

For a vertex v of a graph G, Link(v) is defined to be Link$(K(G, 2))$ and is the graph induced by G on the neighbors of v in G.

For a complex K, K^k is the collection of all simplices of K with dimension less than or equal to k. K^k is called the *k-skeleton* of K. In the case of the 1-skeleton K^1, we also view it as a graph.

A triangulation K of a d-dimensional topological space is called *clean* if $K(K^1, d)$ is homeomorphic to K.

For triangulations of 2-complexes, being clean is equivalent to saying that not only is each face a triangle but also each triangle (of the 1-skeleton) is a face. If a triangulation that is not clean is given, we need only take a subdivision (see below) to ensure that there are no longer any triangles in the 1-skeleton that are not faces in the complex. Hence, every 2-complex admits a clean triangulation.

Example 2. Clean triangulations are nice because we can use them to characterize various types of complexes just by looking at the Link(v) of each vertex v of its 1-skeleton.

- 0-*Dimensional complexes.* All links are void.
- 1-*Dimensional complexes.* Each link is an empty graph or void.
- *Purely* 1-*dimensional complexes.* Each link is an empty graph.
- *Closed surfaces.* Connected graph. Each link is a cycle.
- (*Disconnected*) *closed surfaces.* Each link is a cycle.
- *Surfaces with boundary.* Each link is either a cycle or a path.
- *Pseudosurfaces.* Each link is a union of cyles.
- *Pseudosurfaces with boundary.* Each link is a union of cycles and paths.

In order to obtain a clean triangulation it is sometimes necessary to subdivide a given triangulation. We therefore introduce the concept of a subdivision.

DEFINITION 4. The *subdivision* of a complex K' is a complex K with $|K'| = |K|$, and for each simplex s of K there exists a simplex s' of K' such that $s \subseteq s'$.

The definition of subdivision is given for a geometric complex, though a corresponding interpretation for abstract complexes is immediate. Note that this implies the triangles of the original complex have been subdivided to form a number of triangles in the subdivision, namely, those contained in the original triangle. Likewise, the edges have been subdivided, and of course the subdivision is compatible with that on triangles incident with the edge. For an example see the Su functions in Fig. 3. The subdivision process also provides an answer to whether two complexes with homeomorphic carriers are related in any way. In the 2-dimensional case this was answered by Papakyriakopoulos [6], see also Gross and Tucker [2, pp. 97–98].

THEOREM 2.2. *If two 2-complexes K_1 and K_2 have homeomorphic carriers, then they have a common subdivision.*

Purely 2-dimensional complexes are complexes in which each point and line are contained in a 2-dimensional object. A triangulation of such a complex can be given as a collection of triples of points that constitute the triangles of the triangulation, with the understanding that the points and lines of each triangle are also in the complex.

In the next section we will construct a combinatorial description of purely 2-complexes starting from a clean triangulation. We will show that the description obtained is independent of the triangulation used, hence by the above theorem making it a combinatorial invariant under homeomorphism of purely 2-complexes.

3. Combinatorial description of 2-complexes. In this section we will describe a method of decomposing a complex K into a collection of surfaces with boundaries, together with information about how to reconstruct K from those surfaces. This will enable a complete description of K by a labeled graph. We will show that each homeomorphism type of a 2-complex corresponds uniquely to a particular labeled graph.

We begin by describing the algorithm for decomposing a purely 2-complex K.

ALGORITHM 1. *Complex Splitting Algorithm.*

Input: Clean triangulation T of a complex K is given as a set of triples of vertices.

Algorithm:

Stage 1: The triangles are completely dismantled in a new triangulation $\mathrm{Sp}_1(T)$ (see Fig. 1), obtained from T by taking the vertex set

$$\{(v, t) | v \text{ is a vertex of } T \text{ lying in triangle } t\ \}$$

and triangles

$$\{\{(v_1, t), (v_2, t), (v_3, t)\} | t = \{v_1, v_2, v_3\} \text{ is a triangle of } T\}.$$

Stage 2: The second stage of the process is to reidentify vertices that lie on parts of the complex that are locally a simple surface to create a triangulation $\mathrm{Sp}(T)$. The collection of triangles remains the same. Vertices (v, t) and (v', t') will be reidentified if and only if the following two conditions hold (see Fig. 1):

(i) $v = v'$,

(ii) there is a sequence of triangles of T, $t = t_1, t_2, \ldots, t_n = t'$ and vertices $v_1, \ldots, v_{n-1}$, such that the edge $\{v, v_j\}$ is contained in precisely two triangles t_j and t_{j+1}, $j = 1, 2, \ldots, n-1$.

Output: The triangulation obtained is denoted by $\mathrm{Sp}(T)$.

It corresponds to reidentifying copies of vertices where a vertex in its link has valency two, but note that we reidentify vertices at the center of the link. The only vertices of T that

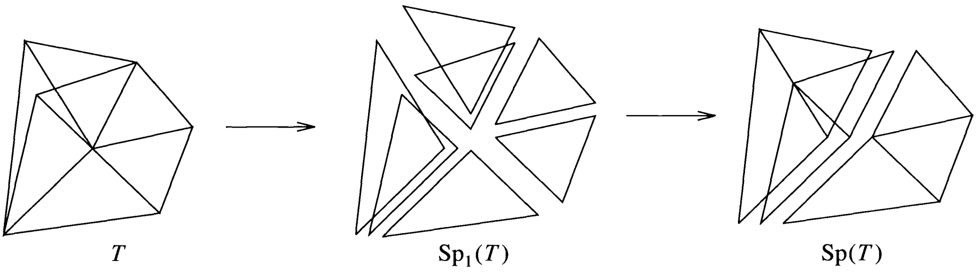


FIG. 1. *Stages in the splitting algorithm.*

are not split in $\mathrm{Sp}(T)$ are those whose Link is a simple circuit. This motivates the following definition.

DEFINITION 5. A vertex of a pure 2-complex is called *simple* if its link consists of a simple circuit. A vertex is called *splitting* if it is not simple.

PROPOSITION 3.1. *After the above splitting operation the complex obtained is a collection of surfaces possibly with boundaries.*

Proof. Consider a vertex (v, t) of the complex created. The vertices in $\mathrm{Link}(v, t)$ in the new complex are

$$\begin{aligned}\{(u, s) \quad | \quad & \exists n \geq 1, \text{ vertices } v_0, \ldots, v_{n-1}, \\ & \text{and triangles } t = t_1, \ldots, t_n = s = \{v, u, v_{n-1}\} \\ & \text{such that for } 1 \leq \ell \leq n-1, \{v, v_\ell\} \text{ lies in precisely } t_\ell \text{ and } t_{\ell+1} \ \}.\end{aligned}$$

Clearly, these vertices form either a path or a circuit, so by the result mentioned in the first section, the complex is a collection of surfaces possibly with boundaries. □

There is, however, a danger that information may be lost in the splitting operation as the next example shows.

Example 3. *Two sphere example.* Consider the triangulation (see Fig. 2)

$$\{\{123\}, \{134\}, \{124\}, \{234\}, \{126\}, \{125\}, \{156\}, \{256\}\}.$$

This is a 2-complex made of two tetrahedrons identified along the edge $\{12\}$. After the splitting operation all but the vertices 1 and 2 are completely reidentified since they are simple vertices. The copies of vertex 1 for triangles $\{156\}$, $\{125\}$, and $\{126\}$ are all reidentified to a vertex that we will denote 1_{56} since vertices 5 and 6 both have degree 2 in Link(1). Similarly the copies of vertex 1 for triangles $\{123\}$, $\{124\}$, and $\{134\}$ are identified to a vertex 1_{34}. Hence, in the split complex there are two copies of vertex 1 (1_{56} and 1_{34}) and similarly two copies of vertex 2 (denoted 2_{56} and 2_{34} using a similar notation, where the indices refer to neighbors in the split complex). So the triangles of the split complex are

$$\begin{gathered}\{\{1_{56}2_{56}6\}, \{1_{56}2_{56}5\}, \{1_{56}56\}, \{2_{56}56\}, \\ \{1_{34}2_{34}4\}, \{1_{34}2_{34}3\}, \{1_{34}34\}, \{2_{34}34\}\},\end{gathered}$$

which is a complex consisting of two disjoint tetrahedra.

The problem with this example is that exactly the same split complex *and* the same linking information obtained from the split vertices would have been obtained from two spheres with just the two anomalous points identified (and no points on the line $\{12\}$ joining them). The type of anomaly of the above example will occur if there are links with only one vertex having

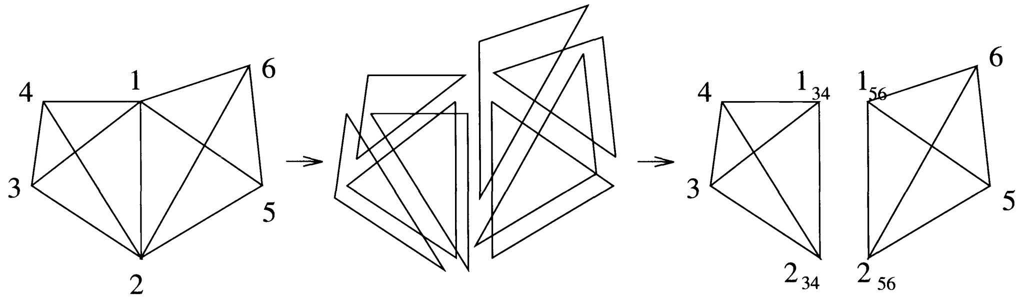


FIG. 2. *Splitting of Example* 3.

valency greater than two and if two such vertices are adjacent in the triangulation. We can remove the problem by taking a subdivision of the complex. In the two spheres example the new vertex on the edge joining the two anomalous vertices would be (two copies of) a split vertex in the case of two spheres joined on a line and two distinct vertices in the case of two spheres joined at two points. The standard pattern for a link is a number of paths between two vertices. This fact motivates the following definition.

DEFINITION 6. A vertex will be termed *standard* if its link consists of a number of paths joining two distinct vertices. Simple vertices are a special case of standard vertices. Nonstandard vertices are called *anomalous* vertices.

We will call a triangulation in which anomalous vertices are adjacent an *anomalous* triangulation.

The problem with anomalous triangulations can, however, be avoided by taking a subdivision of a triangulation in which anomalous vertices are adjacent. To see this is the case we will show that no new vertex of a subdivision is an anomalous vertex. The new vertices lie in the middle of edges or triangles of the old triangulation. For vertices in the middle of an old triangle the new vertex has link a simple circuit and so is not anomalous. For vertices in the middle of an old edge, let the old edge lie on $p \geq 1$ triangles. Therefore, the link of the new vertex consists of p paths between two vertices (the two vertices at the end of the edge of the old triangulation). Hence, the new vertices are not anomalous. We have proved the following lemma.

LEMMA 3.2. *Let $Su(T)$ be a subdivision of a triangulation T. Then $Su(T)$ is not an anomalous triangulation.*

The first stage in our proof that the surfaces and boundary regluing information will characterize the original complex will be to show that homeomorphic complexes give rise to the same collection of surfaces with boundaries after splitting, if we start with nonanomalous triangulations. We begin with a useful lemma.

LEMMA 3.3. *Let $s = \{v_1, v_2, v_3\}$ and $t = \{v_1, v_2, v_4\}$ be two adjacent triangles in a triangulation in which v_1 is a standard vertex. The vertices (v_1, s) and (v_1, t) will be reidentified in the split complex if and only if the edge $\{v_1, v_2\}$ is contained only in s and t.*

Proof. Since v_1 is a standard vertex, its link consists of a number of paths joining two distinct vertices. If $\{v_1, v_2\}$ is contained only in s and t, then by the definition of the second stage of the splitting algorithm, Algorithm 1, the vertices (v_1, s) and (v_1, t) will be reidentified in the split complex. Suppose now that (v_1, s) and (v_1, t) are reidentified in the second stage of the splitting algorithm. If v_2 has degree greater than 2, then v_3 and v_4 must be on distinct paths joining the two vertices of degree greater than 2 in Link(v_1). It is also possible that one of v_3 or v_4 is the second vertex with degree greater than 2. In either case there will be no path involving only vertices of degree two from v_3 to v_4 in Link(v_1) contradicting the reidentification. We conclude that the edge $\{v_1, v_2\}$ is contained only in s and t. □

PROPOSITION 3.4. *For a nonanomalous triangulation of a pure 2-complex the splitting operation commutes with subdivision.*

Proof. Let T be a triangulation of a complex K with no adjacent pair of vertices both anomalous. Let $\mathrm{Su}(T)$ be a subdivision of T. Hence, we must show that $\mathrm{Sp}(\mathrm{Su}(T)) = \mathrm{Su}(\mathrm{Sp}(T))$, where it is understood that the same subdivision is performed on the triangles of $\mathrm{Sp}(T)$ as was performed on T (see Fig. 3).

Since triangles are preserved by the splitting operation, there is clearly a natural mapping τ from the triangles of $\mathrm{Sp}_1(\mathrm{Su}(T))$ to the triangles of $\mathrm{Su}(\mathrm{Sp}(T))$; see Fig. 3. Further, all the vertices in the triangles of $\mathrm{Sp}_1(\mathrm{Su}(T))$ are distinct, so τ induces a mapping

$$\mathrm{vert} : V(\mathrm{Sp}_1(\mathrm{Su}(T))) \to V(\mathrm{Su}(\mathrm{Sp}(T))),$$

where $V(T)$ is the set of vertices of a complex T. To complete the proof we must show that for $(u, s), (u, t) \in V(\mathrm{Sp}_1(\mathrm{Su}(T)))$, $\mathrm{vert}(u, s) = \mathrm{vert}(u, t) \Leftrightarrow (u, s)$ and (u, t) are identified in the second stage of Algorithm 1, i.e., in $\mathrm{Sp}(\mathrm{Su}(T))$. This will imply that the mapping vert induces a triangle-preserving bijection between $\mathrm{Sp}(\mathrm{Su}(T))$ and $\mathrm{Su}(\mathrm{Sp}(T))$.

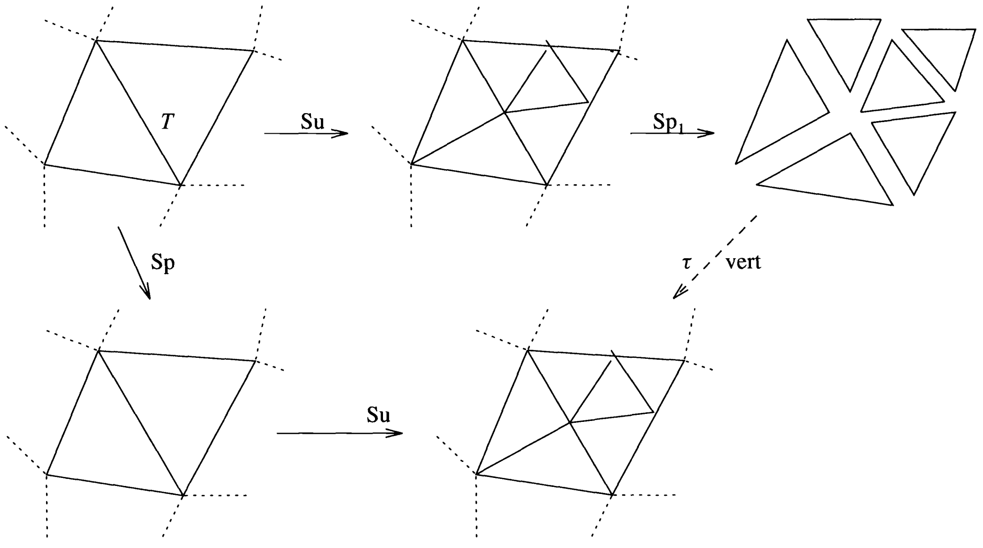


FIG. 3. *Subdivision and splitting in Proposition* 3.5.

CLAIM. *For* $(u, s), (u, t) \in V(\mathrm{Sp}_1(\mathrm{Su}(T)))$, $\mathrm{vert}(u, s) = \mathrm{vert}(u, t) \Leftrightarrow (u, s) \equiv (u, t)$ *in* $\mathrm{Sp}(\mathrm{Su}(T))$.

($\Leftarrow$) First observe that it will be sufficient to show that the property holds for (u, s), (u, t), where s and t are triangles sharing a common edge $\{u, v\}$ in $\mathrm{Su}(T)$, since the full reidentification is the transitive closure of this type of identification. If s and t both lie in the same triangle of T, the edge $\{u, v\}$ lies only in s and t, so (u, s) and (u, t) are identified. But in this case $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$ as required. Suppose therefore that s and t lie in distinct triangles s' and t' respectively of T. The edge $\{u, v\}$ is therefore contained in an edge $\{u', v'\}$ that lies in both s' and t'. If u is a standard vertex and (u, s) and (u, t) are identified, then by Lemma 3.3 $\{u, v\}$ lies only in s and t. Hence, $\{u', v'\}$ lies only in s' and t'. This means that (u', s') and (u', t') are identified in $\mathrm{Sp}(T)$. Likewise, (v', s') and (v', t') are identified and so the copy of the vertex u associated with s in $\mathrm{Su}(\mathrm{Sp}(T))$ is the same as that associated with t, giving $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$ as required. Consider finally the case when u is an anomalous vertex. In this case, by the comment after Definition 6, it must be a vertex of the triangulation T. Again assume that s (t) lies in a triangle s' (t') of T. If (u, s) and (u, t) are identified,

$$\exists s = t_1, t_2, \dots, t_n = t \text{ and } v_1, \dots, v_{n-1},$$

such that $\{u, v_i\}$ lies in precisely two triangles t_i, t_{i+1}. Let the triangle t_i be contained in the triangle t_i' of T. Note that for some i we may have $t_i' = t_{i+1}'$. Where $t_i' \neq t_{i+1}'$, the two triangles meet in an edge $\{u, v_i'\}$ containing $\{u, v_i\}$ and contained only in t_i' and t_{i+1}'. We therefore have a sequence of triangles that indicates (u, s') and (u, t') are identified in $\mathrm{Sp}(T)$. Hence, $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$, as required.

($\Rightarrow$) We consider the case now when two vertices (u, s) and (u, t) in $\mathrm{Sp}_1(\mathrm{Su}(T))$ are not identified in $\mathrm{Sp}(\mathrm{Su}(T))$. We must show that in this case $\mathrm{vert}(u, s) \neq \mathrm{vert}(u, t)$. By the argument at the beginning of the proof of the claim, the fact that the two vertices are not identified implies that s and t must lie in distinct triangles s' and t' of T. We will assume that $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$ and obtain a contradiction. We distinguish two cases according to whether u is a vertex of T or not. If u is a vertex of T, $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$ implies (u, s) and (u, t) are the same vertex in $\mathrm{Su}(\mathrm{Sp}(T))$, which implies that (u, s') and (u, t') were identified in $\mathrm{Sp}(T)$. But this implies the existence of a sequence of triangles $s' = t_1', \dots, t_k' = t'$ with t_i' and t_{i+1}' sharing a common edge containing u and contained only in t_i' and t_{i+1}'. We can therefore find a sequence of triangles

$$s = t_{1,1}, \dots, t_{1,n_1}, t_{2,1}, \dots, t_{k-1,n_{k-1}}, t_{k,1}, \dots, t_{k,n_k} = t$$

with $t_{i,j}$ contained in t_i' and each adjacent pair in the sequence sharing a common edge containing u and unique to the pair in $\mathrm{Su}(T)$. This implies that (u, s) and (u, t) are identified in $\mathrm{Sp}(\mathrm{Su}(T))$, a contradiction.

Finally consider the case when u is not a vertex of T. We must have u contained in an edge $\{u', v'\}$ of T since s and t are in distinct triangles of T. We again assume $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$ and obtain a contradiction. The edge $\{u', v'\}$ of the triangles s' and t' must have been identified in $\mathrm{Sp}(T)$, or we would not have $\mathrm{vert}(u, s) = \mathrm{vert}(u, t)$. But this implies that both (u', s'), (u', t') and (v', s'), (v', t') were identified. Since T does not have adjacent anomalous vertices, we may assume without loss of generality that u' is a standard vertex. By Lemma 3.3, the edge $\{u', v'\}$ is contained only in s' and t'. Now we may find a sequence of triangles

$$s = s_1, \dots, s_\ell, t_1, \dots, t_k = t,$$

with s_i (t_i) contained in s' (t') and each adjacent pair sharing a common edge unique to them which contains u. For all but s_ℓ and t_1 this follows as the two triangles are contained in the same triangle of T, while for s_ℓ and t_1 the common edge is contained in $\{u', v'\}$, which is common to only s' and t'. We conclude that (u, s) and (u, t) are identified, a contradiction. This completes the proofs of the claim and the proposition. □

COROLLARY 3.5. *The collection of surfaces obtained by the splitting operation are invariant under subdivision of the original complex provided it was not anomalous.*

Proof. By the proposition the subdivision operation commutes with the splitting operation when the original complex is not anomalous. But we know that the collection of surfaces is invariant under subdivision, so if we apply the subdivision first (i.e., to the original complex) and then perform the splitting operation, we do not affect the collection of surfaces that is obtained. □

COROLLARY 3.6. *Any two homeomorphic 2-complexes K_1 and K_2 with any nonanomalous triangulations T_1 and T_2 respectively give rise to the same collection of surfaces with boundaries after splitting.*

Proof. Since K_1 and K_2 are homeomorphic, there is a subdivision T_1' of T_1 and subdivision T_2' of T_2 such that T_1' and T_2' are isomorphic. But by Corollary 3.5, the collection of surfaces created by splitting is invariant under subdivision, so T_1 and T_2 give rise to the same collection. □

Each component surface created by splitting a triangulation T corresponds uniquely to some closed (compact) orientable or nonorientable surface with a finite number of holes with boundaries.

ALGORITHM 2. *Graph Construction Algorithm.*
Input: The split triangulation $\mathrm{Sp}(T)$ of a triangulation T together with the triangulation T.
Algorithm: We create a labeled graph $G(T)$ to represent how components of $\mathrm{Sp}(T)$ arose from T.

Stage 1: The component surfaces of the triangulation $\mathrm{Sp}(T)$ are classified according to genus, orientability type, and number of boundary components. Each of the component surfaces will be represented in $G(T)$ by a vertex (vertices of layer one) labeled with the genus and orientability.

Stage 2: The vertices of layer 2 of $G(T)$ will consist of vertices for each vertex of $\mathrm{Sp}(T)$ that arose from a splitting vertex. These vertices are interconnected if they are adjacent in $\mathrm{Sp}(T)$ and are connected to the vertex for the component in which they lie. Hence each boundary of a surface will correspond to a circuit of vertices all adjacent to the surface's vertex in the graph.

Stage 3: Finally there is a layer of vertices corresponding to each splitting vertex of the original complex. These vertices are connected to all the vertices in layer 2 that arose from them.
Output: The labeled graph $G(T)$.

For a vertex v of layer two we will denote by $\mathrm{sib}(v)$ the set of vertices in layer two connected to the same vertex in layer three, that is, the set of vertices arising from the same splitting vertex.

LEMMA 3.7. *Given the triangulation* $\mathrm{Sp}(T)$ *created by the splitting of a triangulation* T, *and the labeled graph* $G(T)$ *defined above, we can reconstruct* T.

Proof. We can clearly reconstruct the vertices of T; they are the vertices of $\mathrm{Sp}(T)$ with identifications performed between those having a common neighbor in the third layer of $G(T)$. As each triangle of T mapped to a unique triangle in $\mathrm{Sp}(T)$, we can also reconstruct the triangles of T once the vertex identification has been made. □

COROLLARY 3.8. *Given only the labeled graph* $G(T)$ *we can reconstruct a complex that is homeomorphic to* T.

Proof. We can construct a triangulation T_0 of the surfaces with boundaries given by the labeled vertices of $G(T)$, which will be homeomorphic to $\mathrm{Sp}(T)$. Moreover, we can ensure that the number of vertices around each boundary corresponds to the number of vertices in the circuit for that boundary in $G(T)$ and that isolated images of splitting vertices in $\mathrm{Sp}(T)$ correspond to vertices of the triangulation T_0. Let $\phi : \mathrm{Sp}(T) \longrightarrow T_0$ be a homeomorphism that maps the boundary vertices and lines as well as the isolated vertices of $G(T)$ to the corresponding vertices and lines of T_0. We can construct a topological space from T_0 using the same procedure as outlined in the lemma, that is, topologically identifying vertices arising from the same splitting vertex and lines arising from the same lines of T. The space created will be a triangulation $T_0^\dagger$ such that $\mathrm{Sp}(T_0^\dagger) = T_0$. Because we have identified closed subsets of points in the topological space, we can lift the homeomorphism ϕ to a homeomorphism $\phi^\dagger : T \longrightarrow T_0^\dagger$. This completes the proof. □

We now introduce a simplification procedure for the graph $G(T)$, which will be shown to give a canonical graph for each homeomorphism class of complex.

ALGORITHM 3. *Simplification Algorithm.*

Input: The graph $G(T)$ arising from a triangulation T.

Algorithm: The following simplification is applied to all vertices of the second layer of $G(T)$ until no change occurs during a complete sweep.

- Given a vertex v of layer two in $G(T)$. For each neighbor u of v with u in layer two, if sib(u) can be paired with sib(v), so that each pair is adjacent in $G(T)$, we identify each of these pairs, the vertices u and v, and their respective neighbours in layer three. The edges from identified vertices are left incident with the new vertex, even if this creates a multiple edge. If two vertices connected by a pair of edges are identified then the new vertex retains a loop.

Output: The simplified graph $\hat{G}(T)$.

The order in which the vertices are scanned does not alter the effect of the algorithm. This is because if a pair of vertices can be merged the merging of any other pair of vertices will not alter this fact (unless of course it corresponds to the same merge) even if one of the vertices merged was from the pair. We will denote by $\hat{G}(T)$ the graph obtained from applying the algorithm to the graph $G(T)$ of a triangulation T.

PROPOSITION 3.9. *Let $\{u, v\}$ be an edge of a nonanomalous triangulation T. Then if T' is the triangulation obtained from T by subdividing the edge $\{u, v\}$, then $\hat{G}(T) = \hat{G}(T')$.*

Proof. First, by Corollary 3.10, subdivision does not affect the collection of surfaces with boundaries that are obtained. Hence, we need only check that the structure of the graphs $\hat{G}(T)$ and $\hat{G}(T')$ are the same.

We consider cases according to the number p of triangles containing the edge $\{u, v\}$. If this number is two then the new vertex v' has Link(v') a simple circuit and so is not a splitting vertex. It therefore does not feature in $G(T')$, which is unchanged from $G(T)$.

If $\{u, v\}$ lies on one triangle, then the new vertex v' has one copy of itself in $G(T')$ adjacent to a copy of u and a copy of v in the boundary of one of the surfaces. If one of u and v only has one copy, then the copy of v' can merge with that vertex under the algorithm given above. We can therefore reduce $G(T')$ to $G(T)$ by one simplification. It follows that $\hat{G}(T) = \hat{G}(T')$. However, w.l.o.g. let v be nonanomalous. Since the edge $\{u, v\}$ lies on one triangle, the vertex u has degree one in Link(v). It therefore follows that Link(v) is a path both in T and T' and therefore that v has only one copy in Sp(T') as required.

Finally consider the case when $\{u, v\}$ lies on $p > 2$ triangles $t_1, \ldots, t_p$. The new vertex v' will split into p copies in Sp(T'). Assume that v is not anomalous. Since the edge $\{u, v\}$ lies on p triangles, u has degree p in Link(v). Therefore, Link(v) is a collection of p paths joining two vertices of degree p both in T and T'. Therefore, the vertex v also splits into p copies in Sp(T'), each of which is adjacent to one of the copies of v' in $G(T')$. It follows that the copies of v and v' can be merged using the above algorithm, again reducing $G(T')$ to $G(T)$ by one simplification. □

COROLLARY 3.10. *If two complexes K_1 and K_2 are homeomorphic and we choose nonanomalous triangulations T_1 of K_1 and T_2 of K_2, then $\hat{G}(T_1) = \hat{G}(T_2)$.*

Proof. Since the triangulations T_1 and T_2 are of homeomorphic 2-complexes, there exists a subdivision T_1' of T_1 and a subdivision T_2' of T_2 such that T_1' and T_2' are isomorphic. By the proposition

$$\hat{G}(T_1) = \hat{G}(T_1') = \hat{G}(T_2') = \hat{G}(T_2),$$

as required. □

Our program of classification of 2-complexes will be complete if we can prove the converse of the above corollary, namely, that given any two complexes K_1, K_2 and nonanomalous triangulations T_1, T_2 of them, such that $\hat{G}(T_1) = \hat{G}(T_2)$, then K_1 is homeomorphic to K_2. This will clearly follow from the following proposition.

PROPOSITION 3.11. *Given a graph $\hat{G}(T)$ arising from a triangulation T, we can construct a topological space V from $\hat{G}(T)$ such that V is homeomorphic to T.*

Proof. In Corollary 3.6 we saw how a complex could be reconstructed from $G(T)$ and that the proof that the reconstructed complex was homeomorphic to T was topological. In this proof we will do the same thing for $\hat{G}(T)$ or rather for each simplification stage of $G(T)$. Let $G(T) = G_1, \dots, G_k = \hat{G}(T)$ be one of the sequences of graphs obtainable when simplifying $G(T)$ to $\hat{G}(T)$. For each G_i, $i = 1, \dots, k$, we will construct a topological space. First, however, take a fixed collection S of surfaces with boundaries and the correct number of isolated points dictated by the layer one structure of the graphs. Note that this does not change in the simplification process since the number of circuits attached to each surface remains the same even when the number of vertices on a circuit is reduced to two or one (a loop will represent a boundary circuit). Now choose the correct number of points on the boundaries of S, one for each vertex in the corresponding boundary circuit of $G(T)$. The surface with the chosen points will be called S_1. Given the surface collection S_i, $i = 1, \dots, k-1$, which will be the surface S with points chosen on the boundary for each vertex of the corresponding circuit in G_i, we construct S_{i+1} from S_i by deleting one of each pair of points corresponding to pairs of vertices that are identified in the simplification of G_i to G_{i+1}. We will prove that we can perform the topological identification of the points and lines of S_i that are identified by the third layer structure of G_i to create a topological space V_i and, furthermore, that it is homeomorphic to T.

By Corollary 3.6 V_1 is homeomorphic to T, while we will take V to be V_k. Note that the construction of V_k is not affected by our knowledge of $G(T)$; so if we prove this result, it will follow that whatever S_k we chose, the resulting space V will be homeomorphic to T. We will proceed by induction. The base case is Corollary 3.6.

Suppose that we have constructed the space V_{i-1}. Let u_j, v_j, $j = 1, \dots, m$, be the pairs of vertices identified as mergeable in G_{i-1} to give G_i, and suppose w.l.o.g. that those corresponding to u_j were deleted from S_{i-1}. Let w_j be the other neighbor of u_j on its boundary (note that w_j could in some cases be v_j). Since the lines of S_{i-1} could be identified to create the space V_{i-1}, all of the u_j, v_j, and w_j were identified amongst themselves, and the collections of edges

$$\{\{u_j, w_j\}\}_{j=1}^m \quad \text{and} \quad \{\{u_j, v_j\}\}_{j=1}^m$$

were also identified. But then the edges $\{v_j, w_j\}$ of S_i can also be identified by the same recipe and in such a way that the chosen points u_j are identified amongst themselves. The space V_i created along with the other identifications performed on S_{i-1} is then homeomorphic to V_{i-1}.

This completes the proof. □

4. Proof of main result. We are now ready to prove our main result.

Proof of Theorem 2.1. We are given two 2-complexes and are required to reduce the problem of deciding whether they are homeomorphic to an isomorphism oracle. We first perform a subdivision so that any singular simplices are removed. Singular simplices are those in which two distinct faces are identified. A subdivision will always remove this type of singularity. A further subdivision will ensure that the triangulation is not anomalous as guaranteed by Lemma 3.2. The next stage is to separate the purely 2-dimensional part of the complex from any lower-dimensional simplices. Zero-dimensional simplices that are not part of a one-dimensional simplex can be counted and, provided there are the same number in both complexes, can be ignored. The set of one-dimensional simplices that are not edges of two-dimensional simplices form a number of graphs, which may or may not be connected to two-dimensional simplices. Those that are not connected to any 2-dimensional simplices can be treated as a single graph in both complexes, and after removing 2-valent vertices we may call the graph isomorphism oracle to decide whether they are homeomorphic. This will be a necessary condition for the two complexes as a whole to be homeomorphic.

Finally we consider those component graphs that are connected to 2-dimensional simplices. These are disconnected from the 2-dimensional part of the complex, but the points of attachment are labeled. We are left with a purely 2-dimensional complex, with a number of labeled points. We perform the construction of the combinatorial invariant using Algorithms 1, 2, and 3 described in the previous section with two slight adaptations. The first adaptation concerns the transfer of labeled points into the graph $G(T)$. For such points that are not on a boundary in the split complex an extra vertex is created in layer two of the graph $G(T)$ connected to the vertex in layer one corresponding to its surface. This vertex will for the time being receive the point's label. For points that lie on a boundary the corresponding vertex of layer three receives the point's label.

The second adaptation concerns the simplification algorithm and requires that labeled vertices cannot be concatenated during the simplification process.

Once the simplified graph $\hat{G}(T)$ has been created we connect the graph sections that were removed from the original complex back onto the vertices with corresponding labels in $\hat{G}(T)$ after concatenating any 2-valent vertices in these graph sections. We then call the graph isomorphism oracle to decide if the two graphs are isomorphic. The fact that the vertices of layer one are labeled ensures that any isomorphism will be an isomorphism of the graphs created from the 2-complexes, together with a homeomorphism of the graph sections retained from the original complex. Hence, by the results of the previous section, the graphs will be isomorphic if and only if the original 2-complexes are homeomorphic. It therefore only remains to show that the reduction algorithm can be performed in polynomial time. □

PROPOSITION 4.1. *Algorithms* 1, 2, *and* 3 *run in time polynomial in the size of the triangulation* T.

Proof. Stage 1 of Algorithm 1 consists of a simple manipulation and can clearly be performed in polynomial time. Stage 2 requires examining pairs of vertices which arose from the same original vertex. By initially labeling edges that appear in only two triangles, this computation becomes equivalent to finding the connected components of a graph and is clearly polynomial.

Stage 1 of Algorithm 2 is a standard computation, while Stages 2 and 3 are straightforward translations using only local information about the connectivity of splitting vertices. All three stages clearly involve only polynomial time computations.

During all but the last sweep of the simplification algorithm (Algorithm 3) the number of vertices in the graph is reduced by at least three. Clearly the number of reductions is therefore at most a third of the total number of vertices in the original graph, which is in turn bounded by the number of vertices and triangles in the original triangulation. It remains therefore to show that each sweep of the algorithm requires at most polynomial time. Checking pairs of vertices for concatenation involves looking at all pairs of adjacent vertices in layer two and examining their sibling sets. The matching of the sibling sets can be checked simply because each sibling has only two neighbors in layer two that are candidate siblings for the second vertex. This computation can clearly be performed in polynomial time. The actual vertex deletion is also a local operation and can be performed in constant time. □

5. Conclusions. We have considered the problem of determining whether two 2-complexes are homeomorphic and investigated its position in the complexity hierarchy. Whereas the closely related problem of determining whether two complexes have isomorphic fundamental groups is algorithmically unsolvable, we have shown that the problem of 2-complex homeomorphism is not only decidable but of equivalent complexity to graph isomorphism modulo polynomial time reductions.

From a practical point of view there exist very efficient heuristic algorithms for testing whether two graphs are isomorphic [5]. Our reduction will therefore facilitate efficient testing of 2-complex homeomorphism.

From a theoretical point of view our results suggest that the problem of 2-complex homeomorphism may be of interest in the on-going investigation of the precise status of graph isomorphism in the complexity hierarchy. We feel that future research could profitably consider which variations of 2-complex homeomorphism retain the same complexity status, which become NP-complete, and which, if any, become undecidable. Some results in this area have been obtained recently. For example, Thomassen has shown that deciding the genus of a graph is NP-complete [8]. This would parallel the studies performed on the complexity status of variants of graph isomorphism, but in the case of 2-complex homeomorphism there is the additional attraction that a closely related problem has already been shown to be undecidable.

REFERENCES

[1] K. S. Booth and C. J. Colbourn, *Problems polynomially equivalent to graph isomorphism*, Tech. report CS-77-04, Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, 1979.

[2] J. L. Gross and T. W. Tucker, *Topological Graph Theory*, Wiley-Interscience, New York, 1987.

[3] A. Lubiw, *Some NP-complete problems similar to graph isomorphism*, SIAM J. Comput., 10 (1981), pp. 11–21.

[4] E. M. Luks, *Isomorphism of graphs of bounded valence can be tested in polynomial time*, J. Comp. Sys. Sci., 25 (1982), pp. 42–65.

[5] B. D. McKay, *Practical graph isomorphism*, Congr. Numer., 30 (1981), pp. 45–87.

[6] C. D. Papakyriakipoulos, *A new proof of the invariance of the homology groups of a complex*, Bull. Soc. Math. Grèce, 22 (1943), pp. 1–154.

[7] T. D. Parsons and T. Pisanski, *Graphs which are locally paths*, in Combinatorics and Graph Theory, Banach Center Public., Vol. 25, PWN, Warsaw, 1989, pp. 127–135.

[8] C. Thomassen, *The graph genus problem is NP-complete*, J. Algorithms, 10 (1989), pp. 568–576.

[9] E. F. Whittlesey, *Classification of finite 2-complexes*, Proc. Amer. Math Soc., 9 (1958), pp. 841–845.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 133–153, February 1994

010

ON COLLISION-FREE PLACEMENTS OF SIMPLICES AND THE CLOSEST PAIR OF LINES IN 3-SPACE*

MARCO PELLEGRINI†

Abstract. The problem of detecting efficiently whether a query simplex is *collision-free* among polyhedral obstacles is considered. If n is the number of vertices, edges, and faces of the polyhedral obstacles, and m is the amount of storage allocated for the data structure ($n^{1+\epsilon} \leq m \leq n^{4+\epsilon}$), it is possible to solve collision-free placements queries for simplices in time $O(n^{1+\epsilon}/m^{1/4})$ for any $\epsilon > 0$, where the constants depend on ϵ. In order to solve this problem the authors develop data structures to detect on-line intersections of query half planes with sets of lines and segments.

Some nearest-neighbor problems for objects in 3-space are also considered. Given a set of n lines in 3-space, the shortest vertical segment between any pair of lines is found in randomized expected time $O(n^{8/5+\epsilon})$ for every $\epsilon > 0$. The longest connecting vertical segment is found in time $O(n^{4/3+\epsilon})$. The shortest connecting segment is found in time $O(n^{5/3+\epsilon})$.

Key words. collision-free placements, Plücker coordinates of lines, arrangements, point location, half-space range searching, parametric search, proximity, three-dimensional space

AMS subject classification. 68U05

1. Introduction.

1.1. Collision-free placements of simplices.

Detecting intersections of objects is a basic problem in computational geometry [PS85], [Ede87], [Meh84]. While intersection problems on the plane are more or less solved, for their three-dimensional counterpart fewer results are known. Intersection of *convex polyhedra* in 3-space can be detected in time logarithmic in the total number n of vertices, faces, and edges, using data structures of linear size [CD80], [DK83], [DK90]. Non-convex polygonal objects in 3-space are much harder from a computational point of view. If we have two *terrains* with the same vertical direction, Chazelle et al. [CEGS89a] solve the intersection-detection problem in time $O(n^{4/3+\epsilon})$. In [MS85] the case of one convex and one non-convex polyhedra is considered. If we have two non-convex polyhedra or any set of non-convex polyhedra, we can solve the intersection-detection problem in time $O(n^{8/5+\epsilon})$ [Pel91b].

In this paper we are interested in detecting the intersection of a query simplex among a set of (possibly intersecting) non-convex polyhedral obstacles. We allow preprocessing of the obstacles and we answer on-line intersection queries in sublinear time. For this problem, a simple-minded extension of the approach of [CD80], [DK90] requires testing each obstacle with the query simplex independently. In the worst case we would have a linear number of simplex-obstacle pairs to consider. The intersection-detection results in [CEGS89a, Pel91b] are inherently off-line and do not immediately imply a solution for our on-line query problem.

The solution of this intersection-detection problem allows us to determine efficiently whether a placement of a polyhedral object of constant complexity within polyhedral obstacles is collision-free. Finding collision-free placements for objects is a basic problem for applications in robotics. There are many results for several variations of the free-placement problem (e.g., finding a collision-free path, finding the placement of the largest copy, etc.) for

*Received by the editors April 13, 1992; accepted for publication (in revised form) September 15, 1992. A preliminary version of this paper appeared with the title "Incidence and nearest neighbor problems for lines in 3-space" in the Proceedings of the 8th ACM Symposium on Computational Geometry, Berlin, Germany, 1992. This research was initiated while the author was visiting the International Computer Science Institute in Berkeley.

†Department of Computer Science, King's College, Strand, London WC2R 2LS, United Kingdom (m.pellegrini@oak.cc.kcl.ac.uk).

a polygonal object amidst polygonal obstacles *in the plane* (see [LS87b], [LS87a], [GSS88], [KS88], [HO89], [CK89], [FHS89], [Tol91]; see [SS89] for a survey).

In [Hal91] some motion planning problems for systems with three degrees of freedom are considered. In this case the robot is mapped as a point in a suitable three-dimensional parametric space and the obstacles are mapped as surfaces within this space. Also, recent work of Aronov and Sharir [AS90], [AS92] on the complexity of a cell in an arrangement of triangles in 3-space has applications to this kind of problem.

The main result of the paper is the following (Theorem 7): given a set of polyhedral obstacles with n vertices, edges, and faces, we can build a data structure using m units of storage, with $n^{1+\epsilon} \leq m \leq n^{4+\epsilon}$, such that for any query simplex we can determine in time $O(n^{1+\epsilon}/m^{1/4})$ whether the simplex is collision-free.[1]

To our knowledge no previous sublinear algorithm for collision-free placement queries among several non-convex polyhedra has been known.

In three-dimensional space, free-placement problems have a high intrinsic combinatorial complexity; therefore we tackle the simple problem of building a data structure for free-placement queries. On the other hand, our data structure is powerful because *both* the placement (six degrees of freedom) *and* the shape (as long as it is of constant complexity) of the robot are part of the query.

We consider first a solution which, at the expense of large storage, answers the queries in logarithmic time. Applying the methods in [CSW90], [Mat91], [AM92a] (see also [AS91a]), we obtain a space/query-time trade-off.

In [Pel91b], [AS91a] algorithms for the following problem are presented: given a set of triangles, count (or report) the triangles intersected by any query ray or segment. One of the main building blocks for our collision-free data structure is a solution for the *dual problem*: given a set of segments in 3-space, count (or report) the segments intersected by any query triangle. Dobkin and Edelsbrunner in [DE84] solve the problem of counting the number of segments intersected by a query *plane*, but their approach does not seem to extend to query *half planes*. Our approach is based on reducing the problem to *half-space range searching* and it attains the goal of a data structure for half plane queries. Once we have the solution for half-planes, it easily extended to triangle queries.

1.2. The closest pair of lines in 3-space. Arrangements of lines in 3-space have been studied recently in computational geometry (e.g., [CEGS89a], [Pel90], [Pel91b], [PS92], [AS91a], [dBHO$^+$91], [CEG$^+$90]). Some of the problems concerning lines can be classified as "incidence" or "order" problems. For example, the *relative orientation* of lines in space [CEGS89a] is exploited in [CEGS89a], [Pel91b], [dBHO$^+$91] to solve ray-shooting problems, and in this paper to answer collision-free queries. Problems that involve a metric on the set of lines in R^3 can be classified as "neighbor" problems.

If a robot does not intersect any obstacle, the next natural question is to find the obstacle *closest to the robot.* Thus, we are led to consider "neighbor" problems in 3-space. Neighbor problems for sets of points have been extensively studied in computational geometry [Vai89], [AESW91], [PS85] also in connection with the well-studied *Voronoi Diagrams* (see [Aur91] for a survey). Much less is known about neighbor problems in higher dimensional spaces *for objects which are not points.*

Neighbor problems for polyhedral objects in 3-space such as lines, segments, and polyhedra are important for applications not only in robotics, but also for three-dimensional very large scale integration and computer-assisted design. In these two areas we often need to enforce a minimum separation distance among three-dimensional objects.

[1] All the bounds presented hold for every $\epsilon > 0$ and the multiplicative constants depend on ϵ.

In §§4.2 and 4.3 we describe algorithms for solving a few neighbor problems on lines in 3-space. We are interested in finding the shortest segment connecting two lines in a set of lines, under a variety of constraints. We give algorithms to find the *shortest vertical segment* connecting two elements in a set of lines or segments which runs in $O(n^{8/5+\epsilon})$ expected time. Surprisingly, we prove an expected time bound for finding the *longest vertical segment* that is only $O(n^{4/3+\epsilon})$. We find the shortest segment connecting two lines in a set of n lines in $O(n^{5/3+\epsilon})$ expected time.

All the neighbor problems considered in this paper can be easily solved in $O(n^2)$ time. For some of the problems slightly sub-quadratic algorithms have been known. For the special case of the shortest vertical distance between two sets of edges of non-intersecting polyhedral terrains Chazelle et al. [CEGS89a] give an $O(n^{4/3+\epsilon})$ randomized expected time algorithm. The problem of the minimum vertical separation for segments was solved in [CS88] in time $O(n^{1.99987})$. The method in [CS88] maps a segment into a point in six-dimensional space. Using the representation in [CS88] and the more recent decomposition technique of [CEGS89b], it is possible to obtain a time bound of roughly $O(n^{2-1/9}) = O(n^{1.8889})$.

Independently, Guibas, et al. [Gui91] have considered the problem of finding the closest pair of lines, obtaining an $O(n^{5/3+\epsilon})$ algorithm. More recently, Chazelle et al. [CEGS92] have improved the running time to $O(n^{8/5+\epsilon})$.

The main strategy used in this paper to solve closest pair problems is to build first a data structure to answer closest line queries. In a second stage we transform the method for solving on-line queries into an off-line method by batching the queries. Because the on-line data structures can have a *multilayer structure* [Meh84], we develop in §4.1 a general method called *nested batching technique*, which is of independent interest (for more applications see [Pel92]).

The paper is organized as follows. In §2 we give an overview of the geometric techniques used to obtain the results. In §3 we discuss the problem of detecting efficiently whether a placement of a simplex is collision-free. Section 4 deals with several neighbor problems: finding the shortest and longest vertical segments and finding the shortest segment in a set of lines.

2. Geometric and algorithmic preliminaries. In this section we survey some geometric and algorithmic results which lay the groundwork for the main results of the subsequent sections.

2.1. Arrangements. A finite set H of hyperplanes in R^d defines a decomposition of R^d into cells of various dimensions, which is called the *arrangement* $\mathcal{A}(H)$ of H [Ede87]. If $|H| = n$, the maximum number of cells in $\mathcal{A}(H)$ is $O(n^d)$ and the arrangement $\mathcal{A}(H)$ can be computed in optimal $O(n^d)$ time [EOS86]. One d-dimensional cell of $\mathcal{A}(H)$ is bounded by $O(n^{\lfloor d/2 \rfloor})$ cells of any dimension [Ede87].

Similarly, a set V of real semi-algebraic varieties in R^d decomposes R^d into cells of various dimensions, which is called the *arrangement* $\mathcal{A}(V)$ of V [CEGS89b]. In [CEGS89b], Chazelle et al. describe a method for building a point-location data structure for the arrangement of n algebraic varieties of fixed maximum degree in R^d. For $d \geq 3$, the data structure has size $O(n^{2d-3+\epsilon})$ and it can be built in $O(n^{2d-3+\epsilon})$ randomized expected time or $O(n^{2d+1})$ deterministic time. The query time is $O(\log n)$.

2.2. Random sampling and cuttings. Given a random sample R of a set of hyperlanes H, with $|R| = r \leq n$, let us consider the arrangement $\mathcal{A}(R)$. A triangulation $\Delta\mathcal{A}(R)$ is a subdivision of each cell of $\mathcal{A}(R)$ into simplices such that the vertices of each simplex are vertices of $\mathcal{A}(R)$. The number of simplices in $\Delta\mathcal{A}(R)$ is $O(r^d)$. The random sampling theory of Clarkson [Cla87] states that with probability at least 1/2 the interior of each simplex

$s \in \Delta\mathcal{A}(R)$ does not meet more than $O(n/r \log r)$ hyperplanes of H. A set as $\Delta\mathcal{A}(R)$ is called a *cutting* for H.

Given H, we can build a data structure that uses $Cn^{d+\epsilon}$ storage, for each $\epsilon > 0$, where the constant C depends on ϵ, such that a query point is located in $\mathcal{A}(H)$ in $O(\log n)$ time [Cla87]. This data structure is built in expected time $O(n^{d+\epsilon})$ [Cla87].

For $d = 2$ Matoušek [Mat90] gives a deterministic method that, for a parameter $r < n$, subdivides the plane into $O(r^2)$ triangles in time $O(nr)$ such that the interior of each triangle meets only n/r lines in H.

For a set V of n algebraic varieties of fixed degree in R^d, $d > 2$, Chazelle et al. [CEGS89b] prove that a R^d can be decomposed into $O(r^{2d-3}\beta(r))$ cells of constant descriptive complexity, where $\beta(\cdot)$ is a sub-logarithmic function, and each cell intersects no more than $O(n/r \log r)$ surfaces in V.

These results on cuttings are the basis of many divide-and-conquer algorithms in computational geometry (see [Aga91] for a survey).

2.3. On-line and off-line point location. The algorithms of [Cla87] and [CEGS89b] solve, in logarithmic time, on-line point location problems in arrangments of hyperplanes or algebraic varieties of fixed degree. In several applications the queries are given off-line and we can batch them in order to speed up the overall computation.

In [EGS90], Edelsbrunner et al. give a method for batched point-location that is generalized in §4.1 to deal with a vast class of off-line point location problems. Given n surfaces and m points, we compute properties of the points that depend only on the locations of these points with respect to the arrangement of the given surfaces in expected time $O(m^{a/(a+1)}n^{a/(a+1)} + m^{1+\epsilon} + n^{1+\epsilon})$, where a is a characteristic parameter of the problem.

To apply this technique an important requirement is the ability to map data and queries as point or surfaces, via dual transformations.

Geometric dual transformations [Ede87], [EMP$^+$82] are pairs of functions: one function maps points to surfaces (hyperplanes) and the second function maps surfaces (hyperplanes) to points in R^d. Duality mappings preserve incidence and order relations.

2.4. Halfspace range searching. A problem intimately connected to the point-location problem is the half-space range searching problem. Given a set S of n points in R^d, build a data structure such that, for every query half-space h^+, the number of points in $S \cap h^+$ is computed efficiently. This problem is solved in [CSW90], [Mat91] using *partition trees*. In a partition tree, each node is associated with a region in R^d such that only a fraction of the children intersect the hyperplane h supporting the query half-space. During the query we retrieve the number of points of S within the regions associated with children not intersected by h and we recurse the query on the children intersected by h. Partition trees are quite versatile and they can be used to set up multilevel data structures.

2.5. Multilevel data structures. Multilevel data structures are a basic paradigm in computational geometry [Meh84]. They are used to search for elements satisfying a complex property. Usually the complex property is split into elementary properties and each elementary property is tested at a specific level of the data structure. For example, in [CSW90], [Mat91] sets of points are organized in multilevel partition trees to answer simplex range queries, where each level of the data structure tests the position of the data points with respect to the hyperplane spanning a facet of the simplex. We have this fundamental theorem in [CSW90].

THEOREM 1 ([CSW90]). *Simplex range searching in n points in R^d can be performed in $O(n^{1+\epsilon}/m^{1/d})$ query time, for every $\epsilon > 0$, using a data structure of size m (for any m between n and n^d), which can be computed in $O(m^{1+\epsilon})$ randomized expected time.*

Matoušek in [Mat91], using a different partition scheme, is able to make the preprocessing deterministic and to reduce the query time to $O(n \log^{O(1)} n/m^{1/d})$.

2.6. Parametric search. In some of the algorithms for nearest neighbor problems on lines we use Megiddo's parametric search technique [Meg83], [Col87]. Roughly speaking, this technique transforms an algorithm to test a property into an algorithm to find the minimum value of a parameter for which the property is true, under quite general conditions.

More formally, suppose we have an algorithm P' to compute a predicate $P(i, t)$, which depends on an input i and a real parameter t. The algorithm P' only uses the parameter t to perform branching tests based on the evaluation of fixed degree polynomials that depend on i and t. Moreover, suppose that the predicate $P(i, t)$ is monotone in t, meaning that for increasing values of t, it never switches from false to true.

Megiddo's parametric search technique is a method that transforms P' into an algorithm P'' for finding the minimum value of t for which the predicate $P(i, t)$ is true.

As an intermediate step of the transformation, we need a parallel version of the program P', which we call $\mathcal{P}$. The parallel time is $T_{\mathcal{P}}$ and the number of processors is p. The new algorithm P'' simulates sequentially $\mathcal{P}$ without specifying the value of t. When each of the p processors is stopped at a branching step that requires the evaluation of a polynomial in t, we compute all the roots of the polynomials at the branching steps and we sort them. Then we use the sequential algorithm P' to perform a binary search on the sequence of roots in order to find the interval where the minimum value t^* lies. Once we have this interval, we can compute the sign of the branching polynomials at t^*. The algorithm branches accordingly to these values. The algorithm completes the simulation and determines a final interval whose left endpoint is exactly t^*. The total time of the simulation is given by $O(pT_{\mathcal{P}} + T_{P'}T_{\mathcal{P}} \log p)$.

Applications of Megiddo's parametric search to geometric problems are in [AASS90], [AM92b], [CEGS92].

2.7. Plücker coordinates of lines. To solve problems on lines and polyhedra we use the *Plücker coordinates* of lines. Algorithmic uses of Plücker coordinates can be found in [CEGS89a] and in [Pel91a], [PS92]; a classical treatment of Plücker coordinates can be found in [Som51].

A point in real three-dimensional space has Cartesian coordinates (x, y, z) and homogeneous coordinates (x_0, x_1, x_2, x_3). The relations between the two systems of coordinates are given by the following equations: $x = x_1/x_0$, $y = x_2/x_0$, and $z = x_3/x_0$. Two points $x = (x_0, x_1, x_2, x_3)$ and $y = (y_0, y_1, y_2, y_3)$ in three-dimensional homogeneous coordinates define a line l in 3-space. The six quantities

$$\xi_{ij} = x_i y_j - x_j y_i \quad \text{for} \quad ij = 01, 02, 03, 12, 23, 31$$

are called *Plücker coordinates* of the line l (oriented from x to y). They correspond to the two-by-two minors of the two-by-four matrix formed by the coordinates of the point x (on the first row) and y (on the second row).

The six parameters are not independent; they must satisfy the following equation (whose solution set constitutes the Plücker hypersurface of Klein quadric or Grassman manifold $\mathcal{F}_4^2$ [Sto89], [Som51]):

$$(1) \qquad \prod \; : \; \xi_{01}\xi_{23} + \xi_{02}\xi_{31} + \xi_{03}\xi_{12} = 0.$$

The incidence relation between two lines l and l' can be expressed using the Plücker coordinates of l and l'. Let a_1, b_1 (respectively, a_2, b_2) be two points on l (respectively, l') oriented as l (respectively, l'). The incidence between l and l' is expressed as the vanishing of the determinant of a four-by-four matrix whose rows are the coordinates of a_1, b_1, a_2, b_2 in this order from top to bottom:

$$\text{(2)} \qquad \begin{vmatrix} a_{10} & a_{11} & a_{12} & a_{13} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ b_{20} & b_{21} & b_{22} & b_{23} \end{vmatrix} = 0.$$

If we expand the determinant according to the two-by-two minors of the sub-matrix formed by the coordinates of the points a_1, b_1 and the minors of the sub-matrix formed by the points a_2, b_2, we obtain the following equation in which only Plücker coordinates are involved:

$$\text{(3)} \qquad \xi_{01}\xi'_{23} + \xi_{02}\xi'_{31} + \xi_{03}\xi'_{12} + \xi'_{01}\xi_{23} + \xi'_{02}\xi_{31} + \xi'_{03}\xi_{12} = 0.$$

Let us introduce two mappings: $\pi \; : \; l \to \pi(l)$ maps a line in $\mathcal{R}^3$ to a hyperplane in $\mathcal{P}^5$ (five-dimensional oriented projective space) whose plane coordinates are the Plücker coordinates of l appropriately reordered. $p \; : \; l \to p(l)$ maps a line in $\mathcal{R}^3$ to a point in $\mathcal{P}^5$ whose coordinates are the Plücker coordinates of the line. The incidence relation between the two lines l, l' (expressed by (3)) can be reformulated as an incidence relation between points and hyperplanes in $\mathcal{P}^5$. Equation 3 can be rewritten in the form $\pi_l(p_{l'}) = 0$, which is equivalent to requiring point $p(l')$ to belong to hyperplane $\pi(l)$. Computations that are standard in real spaces can be done in oriented projective spaces using a method in [Sto89].

For any given pair of lines l and l' the sign of the quantity $\pi_l(pl')$ is called the *relative orientation* of l and l', denoted by $l \diamond l'$.

2.8. Lines in space. In a seminal paper [CEGS89a], Chazelle, et al. use Plücker coordinates to obtain several results on problems involving polyhedral terrains in 3-space.

One of the problems discussed in [CEGS89a] is the following: given a set $\mathcal{L}$ of n lines in $\mathcal{R}^3$, build a data structure so to answer efficiently whether a query line l is above every line in $\mathcal{L}$. They give a data structure of size $O(n^{2+\epsilon})$ to answer such queries in time $O(\log n)$. Because this data structure is the point of departure of the present article from previous work, we discuss it in full detail.

Given two non-vertical lines l and l', l is above l' if there exists a vertical line meeting both lines and the intersection with l is above the intersection with l'. The objective is to express the *above* relation in terms of relative orientation. Let us consider the line λ parallel to l and passing through the point $z_\infty = (0, 0, 1, 0)$. The line l' is *above* l if it is "between" l and λ, therefore we require that

$$\text{(4)} \qquad l' \diamond l = -l' \diamond \lambda.$$

Thus, we need to check the consistency of two linear inequalities. Let $\mathcal{L} = \{l_1, \ldots, l_n\}$ be a set of n *oriented lines*. And let λ_i be the line parallel to l_i and through the point z_∞. Given query line l, we first test if it is *consistently oriented* with respect to $\mathcal{L}$, that is if $l \diamond \lambda_i = -1$ for all i. For a line l that is consistently oriented the aboveness test reduces to test membership of its corresponding Plücker point in the intersection of n Plücker halfspaces.

Chazelle et al. also observe that to decide if a line l is consistently oriented it is sufficient to project l and $\mathcal{L}$ on the xy-plane and to test if the projection of l is clockwise with respect to the projections of all lines in $\mathcal{L}$. Thus, there are $2n$ ways to orient the lines in $\mathcal{L}$ consistently such that any l is consistently oriented exactly for one orientation of $\mathcal{L}$.

3. An incidence problem in 3-space.

3.1. Redefining the *above* relation. Our first objective is to redefine the *above* relation between lines in a form more suitable for algorithmic applications. We do so by generalizing in a more algebraic context the steps of the algorithm in §2.8.

Let us fix once and for all an orthogonal reference frame in 3-space with unit vectors $(\vec{i}, \vec{j}, \vec{k})$ forming a positively oriented triple according to the skew rule. And let (x, y, z) be the coordinates of a generic point in this reference frame.

Given a line l, a vector $\vec{v}$, and a line l' in R^3 we say that l is *above* l' in direction $\vec{v}$ (namely $above(l, l', \vec{v}\,)$) if moving l in direction $\vec{v}$ we eventually intersect l'.

When $\vec{v}$ is a vector parallel to the $-\vec{k}$-axis, we are in the situation described in [CEGS89a]. In [CEGS89a] the condition for a line to be above another line was given by the relative orientation of the two lines in Plücker space and by the relative orientation of the projections of the two lines on the xy-plane. Let us define oriented lines by ordered pairs of points in 3-space: $l = (a, b)$ and $l' = (a', b')$; then the relative orientation of the projections of l and l' on the xy-plane is equivalent to testing if the triple $(a - b, a' - b', \vec{v}\,)$ is oriented as $(\vec{i}, \vec{j}, \vec{k})$. This, in turn, is equivalent to computing the sign of the following determinant:

$$tsp(l, l', \vec{v}\,) = sign \begin{vmatrix} a_x - b_x & a_y - b_y & a_z - b_z \\ a'_x - b'_x & a'_y - b'_y & a'_z - b'_z \\ 0 & 0 & 1 \end{vmatrix},$$

which is the triple scalar product of the vectors $a - b$, $a' - b'$, and $(0, 0, 1)$. We can generalize the notion of relative orientation by making the "vertical direction" part of the query. If the vertical direction is specified by a variable vector $\vec{v}$, the relative orientation is still the sign of the triple scalar product of $(a - b, a' - b', \vec{v}\,)$:

$$(5) \qquad tsp(l, l', \vec{v}) = \text{sign} \begin{vmatrix} a_x - b_x & a_y - b_y & a_z - b_z \\ a'_x - b'_x & a'_y - b'_y & a'_z - b'_z \\ v_x & v_y & v_z \end{vmatrix}$$

Considering the signs $(+1, -1)$ as boolean values, we prove the following technical lemma:

LEMMA 1. $above(l, l', \vec{v}\,) = tsp(l, l', \vec{v}\,)\ xor(l \diamond l')$.

Proof. (i) If $\vec{v} = -\vec{k}$, then clearly $tsp(l, l', \vec{v}) = l' \diamond \lambda$, by the discussion in [CEGS89a]. Because l is above l' if and only if the two sides of (4) have different values the lemma is proved.

(ii) If $\vec{v} \neq -\vec{k}$, then we can apply a rigid transformation to our reference frame which takes $-\vec{k}$ into $\vec{v}$. It is easy to check that such transformation does not change the value of $l \diamond l'$ and $tsp(l, l', \vec{v}\,)$; therefore we are back to case (i) and the lemma is proved. □

We say that a line l is above a set $\mathcal{L}$ in direction $\vec{v}$ if it is above every line in $\mathcal{L}$, namely $above(l, \mathcal{L}, \vec{v}\,)$. From Lemma 1 we have

$$(6) \qquad above(l, \mathcal{L}, \vec{v}\,) = \bigwedge_{i=1,n} tsp(l, l_i, \vec{v}\,)\ xor(l \diamond l_i).$$

3.2. Half-plane intersection queries on lines. The determinant in (5) can be expanded in a bilinear form in terms of the two-by-two minors of the last two rows and the one-by-one minors of the first row. Let us define $\pi'(l) = (a_x - b_x, a_y - b_y, a_z - b_z)$ and $p'(l', \vec{v}) = (u_{yz}, u_{zx}, u_{xy})$, where $u_{ij} = (a'_i - b'_i)v_j - (a'_j - b'_j)v_i$.

We can think of $p'(l', \vec{v})$ as a point in oriented projective 2-space $\mathcal{P}^2$ and of $\pi'(l)$ as a hyperplane in the same space. The value of $tsp(l, l', \vec{v})$ is just the sign of the inner product of $\pi'(l)$ and $p'(l', \vec{v})$. The value of $tsp(l, l', \vec{v})$ is also the sign of the polynomial defining $\pi'(l)$ evaluated at $p'(l', \vec{v})$.

THEOREM 2. *Given a set $\mathcal{L}$ of n lines, there is a data structure $D(\mathcal{L})$ of size $O(n^{2+\epsilon})$ that in time $O(\log n)$, for a query pair $(l', \vec{v})$, determines if l' is above $\mathcal{L}$ in direction $\vec{v}$.*

Proof. We number the lines in $\mathcal{L}$ from 1 to n obtaining a sequence of lines $L = [l_i]$. We construct the arrangement $\mathcal{A}'(\mathcal{L})$ of the hyperplanes in $\{\pi'(l) | l \in \mathcal{L}\}$. Consider a cell c of $\mathcal{A}'(\mathcal{L})$ and a point $q \in c$. We define as the *sign sequence* of the cell c the sequence $[\text{sign}(\pi'(l) \cdot q) | l \in L]$. We build for each region c a Plücker polytope using the negated sign sequence c as a guide for choosing the correct side of the Plücker hyperplanes.

Let $(l', \vec{v})$ be a query pair. First we locate the cell c in the arrangement $\mathcal{A}'(\mathcal{L})$ containing $p'(l', \vec{v})$. Then, we test for inclusion of the Plücker point $p(l')$ in the Plücker polyhedron associated with c.

In order to obtain a space-efficient data structure, we use a a plane partition approach to construct the arrangement in $\mathcal{P}^2$. Using the method in [Mat91], we partition the plane into $O(r^2)$ regions such that each region is cut by no more than n/r hyperplanes $\pi'(l)$. For at most $O(n)$ hyperplanes π' not cutting through a region c, we build a corresponding point-in-polytope query data structure in Plücker space at the expense of at most $O(n^{2+\epsilon})$ storage (see [CEGS89a], [Pel90]). Then we recurse the construction in each region on the hyperplanes cutting through. The storage $S(n)$ needed to construct the data structure satisfies the following inequality:

$$S(n) \leq cr^2 S(n/r) + cr^2 O(n^{2+\epsilon}). \tag{7}$$

The solution is $S(n) = O(n^{2+\epsilon})$. By choosing $r = n^{\nu}$ for an appropriate value of ν, and by adding planar point location data structures, we can achieve $O(\log n)$ query time. Note that the data structure itself does not depend on any specific direction $\vec{v}$. □

COROLLARY 1. *Given a set $\mathcal{L}$ of n lines, there is a data structure $D(\mathcal{L})$ of size $O(n^{2+\epsilon})$ that in time $O(\log n)$, for a query half plane h, determines if h intersects* 0 (*or all*) *lines in $\mathcal{L}$.*

Proof. If a line l is above all lines in $\mathcal{L}$ along a direction $\vec{v}$, then the half plane defined by l and $-\vec{v}$ does not intersect any line in $\mathcal{L}$. Conversely, the half plane defined by l and $\vec{v}$ intersects all lines in $\mathcal{L}$. Here we consider lines parallel to the half plane h as intersecting h (at infinity). □

COROLLARY 2. *Given a set H of n half planes, there is a data structure $D(H)$ of size $O(n^{2+\epsilon})$ that in time $O(\log n)$, for a query line l, determines if l intersects* 0 (*or all*) *half planes in H.*

Proof. A pair $(l', \vec{v})$ determines a half plane h in 3-space. We can think of $p'(l', \vec{v})$ as a hyperplane in $\mathcal{P}^2$ and $\pi'(l)$ as a point in $\mathcal{P}^2$. We build the data structure of Theorem 2 reversing the roles of hyperplanes and points in $\mathcal{P}^2$. For a query line l we can detect efficiently if any line l_i stored in the data structure hits l when moved in the associated direction $\vec{v}_i$. This is equivalent to detecting an intersection of l with all the half planes stored in the data-structure. If a line l hits all the opposite half planes defined by l_i and $-\vec{v}_i$, then it misses all the half planes defined by l_i and $\vec{v}_i$. □

The data structure of Corollary 2 is a generalization of the data structure in [CEGS89a], which can detect intersections of lines and *vertical* half planes only. A data structure solving the same problem of Corollary 2 is in [AM92b].

3.3. Triangle intersection queries on segments. Let S be a set of segments in 3-space. We build a multilevel data structure to solve the following problem: count the number of segments in S intersected by a query triangle. The general strategy is the following. Let t be a query triangle and let $\mathrm{aff}(t)$ be the plane spanning t. The first level of the data structure detects the segments of S which are intersected by $\mathrm{aff}(t)$. Those segments meeting $\mathrm{aff}(t)$ can be extended into full lines without introducing new intersections. The next three levels of the data structure are associated with the three edges of t. For each edge of t we consider the line spanning it and the half plane of $\mathrm{aff}(t)$ based on that line and containing t. We select the lines that meet this query half plane. At the bottom of the multilevel data structures we count exactly the number of segments meeting t. Now we formalize the above strategy and we give bounds on the storage and query time.

THEOREM 3. *Given a set $\mathcal{L}$ of n lines, there is a data structure $D(\mathcal{L})$ of size $O(n^{4+\epsilon})$ that in time $O(\log n)$, for a query half plane h, counts the number of lines in $\mathcal{L}$ intersected by h.*

Proof. We follow the same general approach as in the data structure of Theorem 2. The main difference is that for each planar region we associate a point-location data structure for the whole zone of the Plücker surface in the arrangement of Plücker hyperplanes [Pel91b]. This data structure requires $O(n^{4+\epsilon})$ storage and we can locate a point within the Plücker arrangement in time $O(\log n)$. We prestore in each cell the number of data lines that satisfy the *above* condition with respect to the line supporting the query half plane. We solve the same problem recursively on each planar region relatively to the lines such that $\pi'(l)$ crosses the planar region. The total storage is $O(n^{4+\epsilon})$ and the total query time is $O(\log n)$. □

THEOREM 4. *Given n segments there is a data structure of size $O(n^{4+\epsilon})$ that counts segments intersected by a query triangle in time $O(\log n)$.*

Proof. We dualize every segment in S into a double spatial wedge using a standard point-plane duality $\mathcal{D}$ [Ede87]. Detecting the segments intersected by the plane $\mathrm{aff}(T)$ is equivalent to locate the dual point $\mathcal{D}(\mathrm{aff}(t))$ in the arrangement $\mathcal{A}(\mathcal{D}(S))$ of the wedges.

When we have obtained the set of segments intersected by $\mathrm{aff}(t)$ as the disjoint union of sets stored in the first level of the data structure, we can extend the segments into lines without introducing any new intersection.

We consider now t as the intersections of three half planes in 3-space. In the next three levels of the data structure we store the lines supporting the segments and we build half plane–line intersection data structures of Theorem 3. At the last level, we can count the number of segments intersecting t. The total storage is $O(n^{4+\epsilon})$ as follows from solving a sequence of inequalities similar to (7) and the query time is $O(\log n)$. □

We can also report the intersections in time $O(\log n + k)$, using $O(n^{4+\epsilon})$ storage, where k is the output size.

3.4. Linear storage and sublinear query time. In the previous section we have reduced the problem of counting the number of segments of S intersected by a query triangle to the problem of locating several query points within several arrangements of hyperplanes (Plücker hyperplanes in Plücker space or lines on a parametric plane). By doing so, we compute efficiently, during the query, the relative position of the points with respect to the hyperplanes stored in the data structure.

As a matter of fact we could have interpreted the data structure as storing points and the query as half space range queries. The final result is the same (i.e., checking the sign of polynomials), but the preprocessing is quite different because the data structure can now be constructed using less storage at the expenses of an increased query time.

We can rebuild the whole multilevel data structure of §3.3 using a *half space range searching approach* [CSW90], [AS91a], [Mat91], [AM92a] rather than the point-location approach. The half space range query approaches in [CSW90], [Mat91] use a point location data structure as a subroutine, and in §3.3 we have developed such subroutine for the triangle query problem.

When solving the half space range problem in Plücker space, we have to take into account the presence of the Plücker hypersurface. We have the property that all Plücker points are on the Plücker hypersurface and all Plücker hyperplanes are tangent to the Plücker surface. Adapting the analysis of the data structure in [Mat91] to this special case and using a result in [APS91], Agarwal and Matoušek prove the following lemma:

LEMMA 2 ([AM92a]). *Given n Plücker points it is possible to solve half space range searching problems with Plücker hyperplanes using m storage, with* $n^{1+\epsilon} < m < n^{4+\epsilon}$, $O(m)$ *expected preprocessing time and* $(n^{1+\epsilon}/m^{1/4})$ *query time.*

As in [CSW90], [Mat91] it is possible to trade storage and query time. Using Lemma 2 and the analysis in [AS91a], Agarwal and Matoušek obtain the following result for ray-shooting on a set of triangles in 3-space:

THEOREM 5 ([AM92a]). *Given n triangles in three-dimensional space and any* $\epsilon > 0$, *one can preprocess them in randomized expected time* $O(m)$ *into a data structure of size* $O(m)$, *for* $n^{1+\epsilon} < m < n^{4+\epsilon}$, *so that for a query ray* ρ, *the first triangle met by* ρ *can be computed in time* $O(n^{1+\epsilon}/m^{1/4})$.

Lemma 2 applied to the analysis of §3.3 gives the following theorem:

THEOREM 6. *Given a set of n segments we can build a data structure using m units of storage, with* $n^{1+\epsilon} \leq m \leq n^{4+\epsilon}$, *such that for any query triangle t we can count in time* $O(n^{1+\epsilon}/m^{1/4})$ *the number of esgments intersected by t.*

Theorem 6 is an essential ingredient in the proof of the main theorem of the paper.

THEOREM 7. *Given a set of polyhedral obstacles with n edges, faces, and vertices, we can build a data structure using m units of storage with* $n^{1+\epsilon} \leq m \leq n^{4+\epsilon}$ *such that for any query simplex s we can determine in time* $O(n^{1+\epsilon}/m^{1/4})$ *whether s is collision-free.*

Proof. Assume that we are given a set of polyhedral objects with a total of n edges and a simplex s in 3-space. We have to consider only four collision cases. For each collision case we describe a data structure that answers whether the query simplex s generates a collision of that type.

(i) *An object is completely contained in s.* We select one arbitrary point of each obstacle thus obtaining a set P of at most n points in R^3. We build a data structure D_1 for simplex range searching on P using the methods in [CSW90], [Mat91], which satisfies the stated bound. We use the simplex s to query D_1. If any point of P is in s, then we have a clash of type (i).

(ii) *An edge of s meets a face of same object.* We take the facets of the obstacles and we triangulate them, obtaining a set T of $O(n)$ triangles in 3-space. We build a data structure D_2 for ray-shooting on T using the method in [AS91a] and its improvement in [AM92a]. We take for each vertex v of s the rays from v containing the edges of s incident to v. Using the data structure D_2, we determine the first triangle intersected by each ray. If for some ray the intersection point is closer to v than the second endpoint of the segment spanned by this ray, then we have a collision of type (ii). We have a constant number of ray-shooting queries for each simplex s. The preprocessing and the query time are those stated Theorem 5.

(iii) *A face of s meets an edge of some object.* We take the edges of the obstacles obtaining a set S of $O(n)$ segments in 3-space. We build a data structure D_3 for answering triangle-intersection queries, using the method of Theorem 6. In turns we take the facets of s and we count the number of segments intersected by each facet. If we detect some intersection we have a collision of type (iii).

(iv) *The query simplex s is completely contained within a polyhedral obstacle.* This case is checked by using the data structure D_2 for ray-shooting queries. We take a ray ρ from an arbitrary point of s in an arbitrary direction. If no obstacle is hit the simplex is not contained in any obstacle. Otherwise, we obtain the first facet t hit by ρ. If locally the ray ρ hits t from the interior of the object containing t, then we have a collision of type (iv).

Collision cases (i), (ii), (iii), and (iv) cover all possible collisions between a simplex and a set of polyhedral obstacles. □

4. Neighbor problems for lines in space. To solve neighbor problems on a set of lines in 3-space, we use the following general strategy. Given a set of lines $\mathcal{L}$ we build a data structure $D(\mathcal{L})$ that efficiently answers nearest-line queries for any query line l. The second step is to balance the preprocessing time and the query time by batching the queries, which we know in advance, over the data structure $D(\mathcal{L})$.

On an abstract level we can think that a query is a pair $(l, \mathcal{L})$, where l is any line and $\mathcal{L}$ a set of lines. Solving the query is equivalent to computing the truth value of a conjunction of several predicates that depend on l and on the elements of $\mathcal{L}$. The data structure $D(\mathcal{L})$ allows us to find the truth value of many predicates efficiently in an implicit way and therefore allows us to solve the query problem. Once we have the data structure $D(\mathcal{L})$, we can produce its batched version in a quite general way, which we call *nested batching technique*. This technique is a generalization of an approach to batching computations in [EGS88]. This technique has been used in several recent papers (e.g., [Pel90], [AS91b]) where the underlying predicate has only one conjunct. In [Pel91b] the technique has been used for a formula with two conjuncts. Here we generalize it in an abstract setting for any constant number of conjuncts.

4.1. The nested batching technique. Let $\mathcal{A}$ and $\mathcal{B}$ be two classes of geometric objects and $A \subset \mathcal{A}$ and $B \subset \mathcal{B}$ two finite subsets. Our goal is to compute efficiently $|C_k(A, B)| = |\{(a, b) \in A \times B | F(a, b)\}|$, where $F(a, b) = \wedge_{j=1,k} C_j$ is a conjunction of a constant number k of elementary conditions. Each elementary condition C_j is of the form $S_a^j(p_b^j) > 0$, where $S_a^j(\,.\,)$ is a polynomial whose structure depends on j and $\mathcal{A}$ and whose coefficients depend only on $a \in \mathcal{A}$. Also, p_b^j is a tuple of real numbers depending only on $b \in B$. Moreover, we require that we can rewrite C_j in the dual form $S_b^j(p_a^j) > 0$, where $S_b^j(\,.\,)$ is a polynomial whose structure depends on j and $\mathcal{B}$, whose coefficients depend only on $b \in B$. Also, p_a^j is a tuple of real numbers depending only on $a \in A$. For technical reasons we assume that $C_0 =$ *TRUE*. If $S_a^j(p_b^j) > 0$ we say that the *point* p_b^j is on the *positive side* of the *surface* $S_b^j(\,.\,) = 0$. From now on we will use the terms *point* and *surface* in this context. Note that, for a given j, an element $a \in A$ (respectively, $b \in B$) is mapped to a point or to a surface. We call the point and the surface *dual* to one another.

The following conditions must be also satisfied:

(I) Given a family of surfaces $\{S_a^j(x) = 0 | a \in A\}$ there is a constant $d'(\mathcal{A}, j)$ such that the space underlying the family of surfaces can be partitioned into $M_j'(r) = O(r^{d'(\mathcal{A},j)} \log^{O(1)} r)$ cells of constant descriptive complexity such that each cell meets at most $|A|/r \log r$ surfaces in the family. Moreover, we assume that this decomposition can be computed in expected time $O(|A|r^{O(1)})$.

(II) Given a family of surfaces $\{S_b^j(x) = 0 | b \in B\}$ there is a constant $d(\mathcal{B}, j)$ such that the space underlying the family of surfaces can be partitioned into $M_j(r) = O(r^{d(\mathcal{B},j)} \log^{O(1)} r)$ cells of constant descriptive complexity such that each cell meets at most $|B|/r \log r$ surfaces in the family. Moreover, we assume that this decomposition can be computed in expected time $O(|B|r^{O(1)})$.

Conditions (I) and (II) can be rephrased requiring that there is a $1/r$*-cutting* of a certain size for the arrangement of surfaces (see [Aga91] for a survey on cuttings). To be more precise, we require a cutting covering the portion of space where the query points are confined. If we do not have any condition on the points then the whole space must be covered. If the points are constrained to lie on a surface or within a convex polyhedron, only the surface or the polyhedron must be covered.

Let $d_j = \max\{d'(\mathcal{A}, j), d(\mathcal{B}, j)\}$ and let $d = \max_j d_j$. We will refer to d as the *characteristic dimension of the formula F*, while d_j is the *characteristic dimension of the elementary conjunct C_j*.

Let $T_j(m, n)$ be the time needed to determine the number of pairs $(a, b) \in A \times B$ satisfying the conditions associated with the elementary conjuncts form 1 to j, that is the time to compute $|\mathcal{C}_j(A, B)| = |\{(a, b) \in A \times B| \wedge_{i=1,j} C_i\}|$. We set $M(r) = \max_j\{M_j(r), M'_j(r)\}$. Just to give some examples, for lines and points on the plane we obtain $M'_j(r) = M_j(r) = r^2$ simple regions. For surfaces and points used in §4.3, we obtain $M_j(r) = M'_j(r) = r^4\beta(r)$ simple regions, using the technique in [CEGS89b], where $\beta(r)$ is a sub-logarithmic function. For Plücker points in an arrangement of Plücker hyperplanes we obtain $M_j(r) = M'_j(r) = r^4 \log r$, from a result in [APS91], because we need to cover only the Plücker hypersurface.

LEMMA 3. *Given a set A and B as above, let $|A| = n$ and $|B| = m$. For every $0 \leq j \leq k$, we can build in time $O(n^{d+\epsilon})$ a data structure such that, for every $b \in B$, $|\mathcal{C}_j(\{b\}, A)|$ can be computed in time $O(\log^j n)$.*

Proof. Let $T_j(n)$ be the time needed to build the data structure up to level j. We prove the claim by induction on j. For $j = 0$, $T_0(n) = O(1)$ therefore the lemma is satisfied. Assume $j > 0$. Applying property (I) for a value of r constant from n and m, we compute a decomposition of the space into $M'_j(r)$ elementary cells each one intersected by no more than $n/r \log r$ surfaces in $\{S^j_a = 0|a \in A\}$. For each elementary cell we select a point in its interior and we compute the set S^+_τ of surfaces not intersecting the region τ and in positive position with respect to τ. We associate the cardinality of S^+_τ with the region and we recurse this construction. Also, for each region τ and for all the elements $a \in A$ whose corresponding surface is in S^+_τ, we build the data structure to compute $\mathcal{C}_{j-1}$, which can be built in time $T_{j-1}(n)$. The total time and space for this construction satisfies the recurrence

$$T_j(n) \leq M'_j(r)T_j(n/r \log r) + M'_j(r)T_{j-1}(n) + O(nr^{O(1)} + nM'_j(r)),$$

$$T_j(O(1)) = O(1).$$

Since $M'(r) \leq r^d \log^{O(1)} r$ and assuming by induction hypothesis that $T_{j-1} = O(n^{d+\epsilon})$, the solution to this recurrence is $T_j(n) = O(n^{d+\epsilon})$.

Given a query point p we locate by exhaustive search the region τ containing it. Then we recurse on the two data structures associated with τ. At the last level we collect the counters of the positive regions and we sum them up to form the final result. The query time $Q_j(n)$ satisfies this recurrence:

$$Q_j(n) \leq Q_j(n/r \log r) + Q_{j-1}(n) + O(M'(r)).$$

Assuming by induction hypothesis that $Q_{j-1} = O(\log^{j-1} n)$, we obtain the solution is $Q_j(n) = O(\log^j(n))$. □

From Lemma 3 easily follows Lemma 4.

LEMMA 4. *If $m \geq n^d$ then, for every $j \leq k$, $|\mathcal{C}_j(B, A)|$ can be computed in time $O(m^{1+\epsilon})$.*

Let us suppose now that $m \leq n^d$. We dualize points and surfaces at level j, and we compute the partition (II) for a constant value of r. For each cell τ of the partition we have a set of n_τ points contained it and a set of m_τ surfaces cutting through. By the property (II), $m_\tau \leq cm/r \log r$ for some constant c. For each cell τ the relative sign of the points in τ and the surfaces outside τ is similar for each point in τ. This implies that in time $O(m)$ we can determine for all points in τ and for most surfaces the truth value of the predicate associated with the jth level. For the surfaces intersecting τ we proceed recursively. Let $T_j(m,n)$ be the time needed to compute $|\mathcal{C}_j(B,A)|$. From the above discussion $T_j(m,n)$ satisfies this recurrence:

$$T_j(m,n) \le \sum_{\tau} T_j(m_\tau, n_\tau) + M(r)(m+n) + M(r)T_{j-1}(m,n_\tau) + O(mr^{O(1)}). \tag{8}$$

The following is the main theorem of this section.

THEOREM 8. *For any ϵ and $\delta \le \epsilon/d$,*

$$T_j(m,n) = D_j n^{d/(d+1)+\epsilon} m^{d/(d+1)} + B_j m^{1+\delta} + A_j n \log^{(1+j)} m$$

where D_j, B_j, A_j depend on ϵ and δ.

Proof. The proof is an induction on j and n inspired by similar argument is in [EGS88].

For $j = 0$, $C_0 = \mathit{TRUE}$ is satisfied by nm pairs. The product is computed in constant time. We consider $j > 0$ and we use the inductive hypothesis on T_{j-1}. Equation (8) becomes

$$\begin{aligned} T_j(m,n) \le & \sum_{\tau} T_j(m_\tau, n_\tau) + O(mr^{O(1)}) \\ & + M(r)(D_{j-1}n^{d/(d+1)+\epsilon}m^{d/(d+1)} + B_{j-1}m^{1+\delta} + A_{j-1}n\log^{1+(j-1)} m) \\ & + M(r)(m+n). \end{aligned} \tag{9}$$

For any fixed pair δ, ϵ, with $\delta \le \epsilon/d$ we choose a value of r, constant from n and m, depending on ϵ and δ. The actual dependency of r on ϵ and δ will be defined along with the proof. If $m \ge n^d$ then, from Lemma 4, $T_j(m,n) \le cm^{1+\delta}$ and the theorem is proved assuming $B_j \ge c$. Suppose $m \le n^d$. In this case

$$m^{1+\delta} = m^{d/(d+1)} m^{1/(d+1)+\delta} \le m^{d/(d+1)} n^{d/(d+1)+d\delta}. \tag{10}$$

First notice that at each level of the recursion the term $M(r)n + M(r)A_{j-1}n\log^j m$ in (9) contributes $O(n\log^j m)$ on one level of the recursion; and there are at most $O(\log m)$ levels. This is a consequence of the fact that the objects in A (whose counter is n) are represented as points. The objects in B (which are counted in m) are represented as surfaces. The overall contribution of this term is $O(n\log^{j+1} m)$. It is sufficient to drop this term from the recursion (9) and prove that the modified inequality for a new unknown function $T'_j(m,n)$.

$$\begin{aligned} T'_j(m,n) \le & \sum_{\tau} T'_j(m_\tau, n_\tau) + M(r)m \\ & + M(r)(D_{j-1}m^{d/(d+1)}n^{d/(d+1)+\epsilon} + B_{j-1}m^{1+\delta}) + O(mr^{O(1)}) \end{aligned} \tag{11}$$

satisfies the bound

$$T'_j(m,n) = D_j m^{d/(d+1)} n^{d/(d+1)+\epsilon} + B_j m^{1+\delta}.$$

Choosing B_j and D_j large enough so that the bound holds for small valus of m and n, we apply the inductive argument on $T'_j(m_\tau, n_\tau)$.

$$\begin{aligned} T'_j(m,n) \le & \sum_{i=1}^{M(r)} \left[D_j m_i^{d/(d+1)} n_i^{d/(d+1)+\epsilon} + B_j m_i^{1+\delta} \right] \\ & + M(r)m + M(r)D_{j-1}m^{d/(d+1)}n^{d/(d+1)+\epsilon} + M(r)B_{j-1}m^{1+\delta} + O(mr^{O(1)}), \end{aligned}$$

but $\sum_{i=1,M(r)} m_i^{1+\delta} \le [M(r)c\log^{1+\delta} r/r^{1+\delta}]m^{1+\delta}$. We can sum up terms in m and $m^{1+\delta}$, set $E = B_jM(r)c\log^{1+\delta} r/r^{1+\delta} + M(r) + M(r)B_{j-1} + O(r^{O(1)})$, and use inequality (10). We obtain

$$T_j'(m,n) \le \left[D_j \sum_{i=1}^{M} m_i^{d/(d+1)} n_i^{d/(d+1)+\epsilon} + Em^{d/(d+1)} n^{d/(d+1)+d\delta} \right].$$

Now we eliminate the summation using first the bound on m_i and then the Hölder-Minkowsky inequality [Mit70].

$$\begin{aligned}\sum_{i=1}^{M(r)} m_i^{d/(d+1)} n_i^{d/(d+1)+\epsilon} &\le [(c'\log r/r)m]^{d/(d+1)} \sum_{i=1}^{M(r)} n_i^{d/(d+1)+\epsilon} \\ &\le [c'\log r/r]^{d/(d+1)} m^{d/(d+1)} M(r)^{1/(d+1)-\epsilon} n^{d/(d+1)+\epsilon}.\end{aligned}$$

Assuming $\epsilon \ge d\delta$, we obtain, for some constant c''

$$T_j'(m,n) \le [D_jc''\log^{d/(d+1)} r/r^{d/(d+1)} M(r)^{1/(d+1)-\epsilon} + E]m^{d/(d+1)} n^{d/(d+1)+\epsilon}. \tag{12}$$

Recalling that $M(r) = r^d \log^{O(1)} r$ the coefficient becomes

$$\begin{aligned}&D_jc''\log^{d/(d+1)} r/r^{d/(d+1)} M(r)^{1/(d+1)-\epsilon} + E \\ &= D_jc''\log^{d/(d+1)} r \log^{O(1)/(d+1)-\epsilon} r (1/r)^{d\epsilon} + E.\end{aligned}$$

If we choose r sufficiently large so that the constant in (12) is less than $[D_j/2 + E]$, and if we choose $D_j = 2E$, we obtain

$$T_j'(m,n) \le D_j m^{d/(d+1)} n^{d/(d+1)+\epsilon}$$

which completes the proof. □

It is also easy to show the following corollary.

COROLLARY 3. *It is possible to report the set $C_k(A, B)$ in time $T_k(m,n) + |C_k(A, B)|$.*

COROLLARY 4. *Under the assumption that the cuttings at* (I) *and* (II) *for a constant value of r can be computed in parallel using polylogarithmic time and linearly many processors, then $|C_k(A, B)|$ can be computed in parallel using $O(T_k(m,n))$ processors and polylogarithmic parallel time.*

Proof. The algorithms of Lemma 3 and of Theorem 8 are based on taking a sample of the surfaces and computing a cutting. For each cell in the cutting we find the points within the region, the surfaces intersecting the region and the surfaces on the positive side of the region. Corresponding to every phase of the sequential algorithm we allocate a processor for each pair point-region and for each pair surface-region. In constant time we can determine the input to the next phase. The number of processors we use is bounded by the number of pairs, which in turn is bounded by the sequential time. The parallel time is bounded by the number of phases of the sequential algorithm. This fact, with the hypothesis that we can compute cuttings in polylogarithmic time, gives us a total polylogarithmic query time. □

If we use Clarkson's random sampling techniques [Cla87] to compute the cuttings at (I) and (II), then it is easy to check that constant parallel time and a linear number of processors is sufficient for the construction.

4.2. Finding the shortest vertical segment. We apply the general nested batching technique to the problem of finding the minimum length vertical segment whose endpoints are on two distinct lines of $\mathcal{L}$.

LEMMA 5. *Given a non-vertical line l and a set of non-vertical lines $\mathcal{L}$, the line in $\mathcal{L}$ with the shortest vertical connecting segment to l is the first line hit by l when translated downward or upward.*

Proof. It follows from elementary geometry. □

Our approach to the problem is to find for each line $l \in \mathcal{L}$ the first lines in $\mathcal{L}$ intersected when translating l upward and downward. If l meets more than one line simultaneously, we record only one line. Clearly we produce no more than a $O(n)$ pairs of lines, which we check for the minimum vertical segment in time $O(n)$. In order to find the set of pairs of lines, we use the following theorem from [CEGS89a].

THEOREM 9 ([CEGS89a]). *Given a set $\mathcal{L}$ of n lines in R^3 and any $\epsilon > 0$, we can preprocess $\mathcal{L}$ into a data structure whose storage is $O(n^{2+\epsilon})$, so that for any query line l we can decide whether l is above all lines in $\mathcal{L}$ in $O(\log n)$ time and, if so, find the line $l' \in \mathcal{L}$ hit first by l when translated downward.*

The difference between Theorem 9 and the result discussed in §2.8 is a subtle but important one. The data structure is based on locating Plücker points of the query line in a given Plücker polytope formed by the hyperplanes associated with the data lines in $\mathcal{L}$. The point location data structure is built by taking a random sample of the hyperplanes and by triangulating the interior of the resulting polytope. In order to obtain the result of §2.8 any triangulation would do. For the result of Theorem 9 Chazelle et al. carefully devise a triangulation so that, for every "triangle" produced, any Plücker point in the "triangle" would hit the same line in the random sample when translated downward. During the point location query we collect at most a logarithmic number of candidate data lines. Then in extra logarithmic time we select the data line immediately below the query line.

First we establish the following theorem.

THEOREM 10. *Given a set $\mathcal{L}$ of n lines in R^3 and any $\epsilon > 0$, we can preprocess $\mathcal{L}$ into a data structure whose storage is $O(n^{4+\epsilon})$, so that for any query line l we can find in $O(\log^2 n)$ time the line $l' \in \mathcal{L}$ hit first by l when translated downward (or upward).*

Proof. Using a data structure similar to that of Theorem 3, we can determine for a query line the lines in $\mathcal{L}$ above and below it. For these lines, which are represented as a disjoint union of canonical sets, we apply the result of Theorem 9. We obtain a data structure that for every query line l gives us the line in $\mathcal{L}$ immediately above and the line immediately below. The storage $S(n)$ used by the data structure satisfies the following inequality:

$$S(n) \leq r^4 S(n/r \log r) + r^4 n^{2+\epsilon}$$

from which we have $S(n) = O(n^{4+\epsilon})$. □

4.2.1. The batching technique and the shortest vertical segment.

THEOREM 11. *Given n lines in 3-space it is possible to find the shortest vertical connecting segment in $O(n^{8/5+\epsilon})$ randomized expected time for any $\epsilon > 0$, where the multiplicative constant depends on ϵ.*

Proof. We divide the set of lines $\mathcal{L}$ into two sets L_1 and L_2 of roughly equal size. We solve the problem recursively for the two sets and then we consider the bichromatic problem (i.e., finding the minimum distance between a line in L_1 and a line in L_2). The total complexity is dominated by the solution for the bichromatic case.

We now apply the nested batching technique of §4.1 to the data structure of Theorem 10 (Corollary 3). L_1 is used as our data set of lines and L_2 as the query set for the algorithm of Theorem 10.

At the first level of the data structure, we have to compute the predicate $tsp(l, l', \vec{v})$, where $\vec{v}$ is the vertical direction. The characteristic dimension of this test is $d_1 = 1$ (the vector $\vec{v}$ is fixed). At the second level we locate Plücker points in an arrangement of Plücker hyperplanes. From results in [APS91] we have characteristic dimension for this test $d_2 = 4$. At the last level we use the special cutting of Theorem 9 with characteristic dimension $d_3 = 2$. At the last level, by locating the Plücker points within the regions of the cutting, we are able to find also the closest data line.

By applying the nested batching method to the data structure of Theorem 10, we would like to obtain for each line in L_1 the line in L_2 with the shortest vertical segment. Unfortunately the batched nested method does not quite produce such set of pairs.

At the second level of the algorithm we have sets of Plücker hyperplanes $H(M_\sigma)$ and sets of Plücker points $P(N_\sigma)$ such that the relative orientation is known and uniform (i.e., all lines in M_σ are above all lines in N_σ). By batching the algorithm of Theorem 9 on M_σ as data and N_σ as points, we obtain a set $\mathcal{Q}$ of pairs of lines, where each line in N_σ is associated to the line in M_σ immediately below. But, in order to use the nested batching technique, we must sometimes flip the roles of data and queries. Reversing the role of M_σ and N_σ, we obtain a set of pairs $\mathcal{Q}'$. The two sets $\mathcal{Q}$ and $\mathcal{Q}'$ can be very different one from the other. On the other hand if $(l, l') \in \mathcal{Q}$ is the pair of lines with minimum connecting vertical segment, then (l', l) is an element of $\mathcal{Q}'$ and it is the pair of lines with minimum connecting vertical segment in $\mathcal{Q}'$. Reversing the role of the two sets of lines, we do not loose the information about the minimum connecting segment and the algorithm produces the correct answer. The time bound follows from Theorem 8. □

We now extend the result of Theorem 11 to segments. It is enough to solve the bichromatic version of the problem where we are given two sets of segments R (red) and B (blue). The main idea is to find a set Σ of pairs of sets (R_σ, B_σ) with the following properties:

(a) For every σ, $R_\sigma \subseteq R$ and $B_\sigma \subseteq B$.

(b) Every pair of red, blue segments whose xy-projections intersect is represented in one and only one pair σ.

(c) The xy-projection of every segment in R_σ intersects the xy-projection of every segment in B_σ.

For each pair σ we can safely extend the segments into full lines without introducing new vertical segments. We use the result of Theorem 11 on each pair and pick the minimum among the partial results.

In order to find such set Σ, we need a few auxiliary definitions. Given a segment s on the plane, let l_s be the line supporting s, $rph(s)$ the right endpoint and $lhp(s)$ the left endpoint. $Slope(l)$ is the slope of line l.

LEMMA 6. *Two segments s_1 and s_2 on the plane intersect if and only if $l(s_1)$ meets s_2 and $l(s_2)$ meets s_1.*

LEMMA 7. *A line l meets a segment s in either of these two cases:*

(A) $(Slope(l) < Slope(l_s)) \wedge (lph(s)\ below\ l) \wedge (rhp(s)\ above\ l)$

(B) $(Slope(l) > Slope(l_s)) \wedge (lhp(s)\ above\ l) \wedge (rhp(s)\ below\ l)$

Proof. Follows from elementary geometry. □

COROLLARY 5. *Given n segments in 3-space it is possible to find the shortest vertical connecting segment in $O(n^{8/5+\epsilon})$ randomized expected time for any $\epsilon > 0$, where the multiplicative constant depends on ϵ.*

Proof. We build a multilevel data structure using the nested batching technique in order to find pairs of red-blue segments whose xy-projection satisfies the properties of Lemma 6 and Lemma 7. At the first level we compare the slopes of the segments. This gives us a test of characteristic dimension $d = 1$. Once we have groups of segments whose relative slope is uniform, we can test at the next levels the above/below relations using standard point/line duality in dimension $d = 2$. At the last level we obtain pairs of sets of segments (blue and red) such that all red segments meet all blue segments. We can extend the corresponding red-blue segments in 3-space into full lines without introducing any new and potentially dangerous candidates for the minimum vertical connecting segment test. At the last level we therefore add a data structure similar to those of Theorem 11. The characteristic dimension of the whole data structure is therefore dominated by $d = 4$. The time bound derives from the application of the nested batching technique of Theorem 8. □

4.2.2. Finding the longest vertical segment. Suppose we want to find the *longest* among the vertical segments connecting two lines in $\mathcal{L}$. Although this problem has a formulation similar to finding the *shortest* vertical segment, we obtain an asymptotically faster algorithm.

THEOREM 12. *Given a set $\mathcal{L}$ of n lines in R^3, the maximum length vertical segment connecting two lines in $\mathcal{L}$ can be found in $O(n^{4/3+\epsilon})$ randomized expected time.*

Proof. We partition $\mathcal{L}$ into two subsets, L_1 and L_2, of equal size and we solve the problem recursively in each set. It is thus enough to solve the bichromatic case. Suppose without loss of generality the pair (l_1, l_2), with $l_1 \in L_1$ and $l_2 \in L_2$, achieves the maximum and moreover l_1 is above l_2. Let δ be the value of the maximum. If we shift every line in L_1 downward of a distance δ, we obtain a new set L'_1. From the fact that δ is the maximum distance follows that every line in L'_1 is below every line in L_2. The two sets L'_1 and L_2 satisfy the towering property (see [CEGS89a]), which can be tested in $O(n^{4/3+\epsilon})$ randomized expected time. This algorithm can be seen as a special instance of the nested batching technique of §4.1. In this case we have a first level to test the *tsp* condition with characteristic dimension $d_1 = 1$ and a second level to test the relative orientation with characteristic dimension $d_2 = 2$. The algorithm for testing the towering property can be used as the oracle of Megiddo's parametric search method. A parallel version of the oracle can be devised using the inherent tree-structure of the nested batching scheme of §4.1, as shown in Corollary 4. We obtain logarithmic parallel time using no more than $O(n^{4/3+\epsilon})$ processors. The parametric search approach allows us to find the value of δ that is the maximum bichromatic distance. □

The result of Theorem 12 extends, with the same time bound, to sets of segments, since the characteristic dimension of each level is no more than $d = 2$, as follows from Lemma 6 and 7 and from the algorithm to test the towering property [CEGS89b].

4.3. Finding the shortest segment. Given a set $\mathcal{L}$ of n lines in R^3 the problem is to find the shortest segment connecting two lines. A more formal statement of the problem is the following: compute $\text{Min}(\mathcal{L})$, where

$$\text{Min}(\mathcal{L}) = \min_{l',l\in\mathcal{L},1\neq l'} \min_{p\in l,q\in l'} |\bar{pq}|.$$

Computing $\text{Min}(\mathcal{L})$ fits nicely into the general schema of Megiddo's parametric search. Given a set $\mathcal{L}$ of n lines in R^3 we partition $\mathcal{L}$ into two disjoint sets R (red) and B (blue) of roughly $n/2$ lines each. If $T(n)$ is the time of the algorithm for finding the closest pair, we have the following recursive inequality:

$$T(n) \leq 2T(n/2) + T'(n/2, n/2)$$

where $T'(n, m)$ is the time sufficient to solve the bichromatic version of the problem (i.e., find the shortest segment connecting a "red" line with a "blue" line).

Let us denote as $P'(R, B, t)$ the program that, given a set R of red lines and a set B of blue lines, counts the number of red lines at distance less than t from any blue line. Clearly

$P'(R, B, t)$ is monotone in t. Using a method in [Pel91b], we can easily check if $P'(R, B, 0) = 0$. Finding the shortest bichromatic distance is equivalent to finding the minimum value of t such that $P'(R, B, t) > 0$.

4.3.1. The oracle. In this subsection we give the algorithm underlying the program $P'(R, B, t)$. The problem of counting pairs of lines within distance t can be seen as the problem of counting the number of intersections among the set R of lines and the set B' of cylinders of radius t whose axes are lines in B.

LEMMA 8. *Given a set of n lines and a set of m cylinders of the same radius, there is an algorithm that counts all line-cylinder intersections in time*:

$$Am^{5/6}n^{5/6+\epsilon} + Bm^{1+\epsilon} + Cn\log^2 m,$$

where the constants A, B, and C depend on ϵ.

Proof. A line l in 3-space is representable in a canonical way as the intersection of a plane parallel to the x-axis and of a plane parallel to the y-axis.[2]

$$y = az + b.$$

$$x = cz + d.$$

The four numbers (a, b, c, d) uniquely represent a line in 3-space. Thus we can map a line l to a point $p(l)$ in real parametric space R^4. A cylinder c of fixed radius t is representable in a canonical way as a point $p(c)$ in real parametric space R^4 (as a matter of fact we need only to map the axis of the cylinder).

Given a line l we define as the dual surface of l the set $S(l)$ of all the points $p(c)$ where c is a cylinder of radius t tangent to l. Clearly $S(l)$ is an algebraic variety (of co-dimension 1) in R^4. In particular there is a polynomial $q_l(\cdot)$ such that $S(l) = \{x \in R^4 | q_l(x) = 0\}$. Symmetrically we define as the dual surface of the cylinder c the set $S(c)$ of the points $p(l)$ where l is tangent to c. Again $S(c)$ is an algebraic variety of co-dimension 1. And we know there exists a polynomial $r_c(\cdot)$ such that $S(c) = \{x \in R^4 | r_c(x) = 0\}$. The relative position of the point $p(l)$ and surface $S(c)$ determines whether l meets c or not.

Using the data structure in [CEGS89b], we can locate the points $p(l)$ in the arrangement of algebraic surfaces $S(c)$. The size is $O(n^{2d-3+\epsilon})$. The query time is $O(\log n)$. We apply the batching argument developed in §4.1 for locating m points in an arrangement of n surfaces. The time bound follows from the batching technique. □

THEOREM 13. *Given n lines in 3-space, it is possible to find the shortest connecting segment in $O(n^{5/3+\epsilon})$ randomized expected time for any $\epsilon > 0$, where the multiplicative constant depends on ϵ.*

Proof. The algorithm of Lemma 8 (based on the data structure in [CEGS89b]) and on the nested batching technique has a tree structure of logarithmic depth. It is relatively easy to transform the sequential algorithm into a parallel algorithm using Corollary 4, as required by Megiddo's technique. □

The oracle used in Theorem 13 is a counting oracle that counts the number of intersections. In order to solve the closest line problem it is sufficient for the oracle to test whether the number of incidences is greater than 0. It is likely that such an algorithm can attain a better asymptotic bound than the counting oracle. Such improved bound would imply a faster algorithm for finding the closest line. The improved time bound of [CEGS92] is based on a similar observation.

[2] We cannot represent horizontal lines but, with a suitable space transformation, we can make sure that we do not have any.

5. Conclusions and open problems. Solving problems on lines in 3-space is often a necessary ingredient for solving problems on sets of polyhedra in 3-space. In this paper we solved an on-line intersection query problem for lines and half planes in 3-space, which is an essential ingredient of a fast method for collision-free placement queries amidst polyhedral obstacles in 3-space. In the same spirit, we gave algorithms for solving a few neighbor problems on lines in 3-space. Here are a few open questions: Is it possible to extend the solution of neighbor problems of lines to segments and polyhedra? How can we combine neighbor information and collision-free placements to plan a collision-free movement of a polyhedral object in a polyhedral environment? Is it possible to have a dynamic (under insertions and deletions) version of the data structure of Theorem 7? An initial result in the direction of solving this last question is in [AM91].

6. Acknowledgments. I thank Subhash Suri for proposing the problem of finding the closest pair of lines and Raimund Seidel, Micha Sharir, and Leonidas Guibas for useful discussions. I also thank Pankaj Agarwal for his comments on an early version of the paper and Professor Julian Ullmann for his comments.

REFERENCES

[AASS90] P. K. Agarwal, B. Aronov, M. Sharir, and S. Suri, *Selecting distances in the plane*, in Proc. of the 6th ACM Symposium on Computational Geometry, Berkeley, CA, 1990, pp. 321–331.

[AESW91] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl, *Euclidean minimum spanning tree and bichromatic closest pair*, in Proc. of the 7th ACM Symposium on Computational Geometry, North Conway, NH, 1991, pp. 203–210.

[Aga91] P. K. Agarwal, *Geometric partitioning and its applications*, Tech. report CS-1991-27, Dept. of Computer Sci., Duke University, Durham, NC, 1991.

[AM91] P. K. Agarwal and J. Matoušek, *Dynamic half-space range reporting and its applications*, Tech. report CS-1991-43, Dept. of Computer Science, Duke University, Durham, NC, 1991.

[AM92a] ———, *Range searching with semialgebraic sets*, in Proc. of the 17th Symp. on Mathematical Foundations of Computer Science, in Lecture Notes in Computer Science 629, 1992, pp. 1–13.

[AM92b] ———, *Ray shooting and parametric search*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 517–526.

[APS91] B. Aronov, M. Pellegrini, and M. Sharir, *On the zone of an algebraic surface in a hyperplane arrangement*, Discrete Comput. Geom., 9 (1993), pp. 177–186. Preliminary version in Proc. of the 1991 Workshop on Algorithms and Data Structures and Proc. of the 3rd Canadian Conf. on Comp. Geom., Ottawa, Vancouver, Canada.

[AS90] B. Aronov and M. Sharir, *Triangles in space or building (and analyzing) castles in the air*, Combinatorica, 10(2) (1990), pp. 137–173.

[AS91a] P. K. Agarwal and M. Sharir, *Applications of a new space partitioning technique*, in Proc. of the 1991 Workshop on Algorithms and Data Structures, Ottawa, Vancouver, Canada, Lecture Notes in Computer Science 519, Springer Verlag, Berlin, New York, 1991, pp. 379–391.

[AS91b] ———, *Counting circular arc intersections*, in Proc. of the 7th ACM Symposium on Computational Geometry, Ottawa, Vancouver, Canada, 1991, pp. 10–20.

[AS92] B. Aronov and M. Sharir, *Castles in the air revisited*, in Proc. of the 8th ACM Symposium on Computational Geometry, Berlin, Germany, 1992, pp. 146–256.

[Aur91] F. Aurenhammer, *Voronoi diagrams–a survey of a fundamental geometric data structure*, ACM Computing Surveys, 23(3) (1991) pp. 345–405.

[CD80] B. Chazelle and D. P. Dobkin, *Detection is easier than computation*, in Proceedings of the 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 146–153.

[CEG$^+$90] B. Chazelle, H. Edelsbrunner, L. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink, *Counting and cutting circles of lines and rods in space*, in Proc. of the 31st Annual Symposium on Foundations of Computer Science, 1990.

[CEGS89a] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *Lines in space: combinatorics, algorithms and applications*, in Proc. of the 21st Symposium on Theory of Computing, 1989, pp. 382–393.

[CEGS89b] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir, *A singly exponential stratification scheme for real semi-algebraic varieties and its applications*, in Proc. of the 16th International Colloquium on Automata, Languages and Programming, number 372 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 1989, pp. 179–193.

[CEGS92] ———, *Diameter, width, closest line pair and parametric search*, in Proc. of the 8th ACM Symposium on Computational Geometry, Berlin, Germany, 1992, pp. 120–129.

[CK89] L. P. Chew and K. Kedem, *Placing the largest similar copy of a convex polygon among polygonal obstacles*, in Proc. of the 5th ACM Symposium on Computational Geometry, Saarbrucken, Germany, 1989, pp. 167–174.

[Cla87] K. L. Clarkson, *New applications of random sampling in computational geometry*, Discrete Comput. Geom. 2 (1987), pp. 195–222.

[Col87] R. Cole, *Slowing down sorting networks to obtain faster sorting algorithms*, J. of Assoc. Comput. Mach., 34 (1987), pp. 200–208.

[CS88] B. Chazelle and M. Sharir, *An algorithm for generalized point location and its applications*, Tech. report 153, Robotics Lab., Courant Institute of Mathematical Sciences, New York, 1988.

[CSW90] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, in Proc. of the 6th ACM Symposium on Computational Geometry, Saarbrucken, Germany, 1990, pp. 23–33.

[dBHO$^+$91] M. de Berg, D. Halperlin, M. Overmars, J. Snoeyink, and M. van Kreveld, *Efficient ray-shooting and hidden surface removal*, in Proc. of the 7th ACM Symposium on Computational Geometry, Saarbrucken, Germany, 1991.

[DE84] D. Dobkin and H. Edelsbrunner, *Space searching for intersecting objects*, in 25th FOCS, 1984, pp. 387–392.

[DK83] D. P. Dobkin and D. G. Kirkpatrick, *Fast detection of polyhedral intersection*, Theoret. Comp. Sci. 27 (1983), pp. 241–253.

[DK90] D. Dobkin and D. Kirkpatrick, *Determining the separation of preprocessed polyhedra: a unified approach*, in Proc. of the 17th International Colloqium on Automata, Languages and Programming, 1990, pp. 400-413.

[Ede87] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, New York, 1987.

[EGS88] H. Edelsbrunner, L. Guibas, and M. Sharir, *The complexity of many faces in arrangements of lines and segments*, in Proc. of the 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1988, pp. 44–55.

[EGS90] ———, *The complexity and construction of many faces in arrangements of lines and segments*, Discrete Comput. Geom., 5 (1990), pp. 161–196.

[EMP$^+$82] H. Eedelsbrunner, H. Mauer, F. Preparata, E. Welzl, and D. Wood, *Stabbing line segments*, BIT, 22 (1982), pp. 274–281.

[EOS86] H. Edelsbrunner, J. O'Rourke, and R. Seidel, *Constructing arrangements of lines and hyperplanes with applications*, SIAM J. Comput., 15 (1986), pp. 341–363.

[FHS89] J. Friedman, J. Hershberger, and J. Snoeyink, *Compliant motion in a simple polygon*, in Proc. of the 5th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1989, pp. 175–186.

[GSS88] L. J. Guibas, M. Sharir, and S. Sifrony, *On the general motion planning problem with two degrees of freedom*, in Proc. of the 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1988, pp. 289–298.

[Gui91] L. Guibas, Personal communication, November 1991.

[Hal91] D. Halperlin, *On the complexity of a single cell in certain arrangements of surfaces in 3-space*, in Proc. of the 7th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1991, pp. 314–323.

[HO89] D. Halperlin and M. Overmars, *Efficient motion planning of an L-shaped object*, in Proc. of the 5th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1989, pp. 156–166.

[KS88] K. Kedem and M. Sharir, *An automatic motion planning system for a convex polygonal mobile robot in 2-d polygonal space*, in Proc. of the 4th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1988, pp. 329–340.

[LS87a] D. Leven and M. Sharir, *An efficient simple motion planning algorithm for a ladder moving in two-dimensional space amidst polygonal barriers*, J. Algorithms, 8 (1987), pp. 192–215.

[LS87b] ———, *Planning a purely translational motion for a convex object in two-dimensional space using generalized voronoi diagrams*, Discrete Comput. Geom., 2 (1987), pp. 9–31.

[Mat90] J. Matoušek, *Cutting hyperplane arrangements*, in Proc. of the 6th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1990, pp. 1–9.

[Mat91] J. MATOUŠEK, *Efficient partition trees*, in Proc. of the 7th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1991, pp. 1–9.

[Meg83] N. MEGIDDO, *Applying parallel computation algorithms in the design of sequential algorithms*, J. Assoc. Comput. Mech., 30 (1983), pp. 852–865.

[Meh84] K. MEHLHORN, *Multidimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, New York, 1984.

[Mit70] D. S. MITRINOVIC, *Analytic Inequalities*, Berlin, New York, Springer-Verlag, 1970.

[MS85] K. MEHLHORN AND K. SIMON, *Intersecting two polyhedra one of which is convex*, in Proc. of Fundamentals of Computation Theory, Lecture Notes in Computer Science 199, Springer-Verlag, Berlin, New York, 1985, pp. 534–542.

[Pel90] M. PELLEGRINI, *Stabbing and ray shooting in 3-dimensional space*, in Proc. of the 6th ACM Symposium on Computational Geometry, Urbana-Champaign, IL, 1990, pp. 177–186.

[Pel91a] ———, *Combinatorial and algorithmic analysis of stabbing and visibility problems in 3-dimensional space*, Ph.D. thesis, New York University–Courant Institute of Mathematical Sciences, New York, 1991. (Courant Institute, Robotics Lab. report 241.)

[Pel91b] ———, *Ray-shooting on triangles in 3-dimensional space*, Algorithmica, 9 (1993), pp. 471–494. (Also in Lecture Notes in Computer Science 519, pp. 20–31.)

[Pel92] ———, *A new algorithm for counting circular arc intersections*, Tech. report TR-92-010, International Computer Science Institute, Berkeley, CA, 1992.

[PS85] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, Berlin, New York, 1985.

[PS92] M. PELLEGRINI AND P. SHOR, *Finding stabbing lines in 3-space*, Discrete Comput. Geom., 8 (1992), pp. 191–208.

[Som51] D. M. H. SOMMERVILLE, *Analytical geometry of three dimensions*, Cambridge University Press, Cambridge, England, 1951.

[SS89] J. T. SCHWARTZ AND M. SHARIR, *A survey of motion planning and related geometric algorithms*, in Geometric Reasoning, D. Kapur and J. L. Mundy, eds., MIT Press, Cambridge, MA, 1989, pp. 157–159.

[Sto89] J. STOLFI, *Primitives for computational geometry*, Tech. report 36, Digital SRC, Palo Alto, CA, 1989.

[Tol91] S. TOLEDO, *Extremal polygon containment problems*, in Proc. of the 7th ACM Symposium on Computational Geometry, 1991, pp. 176–185.

[Vai89] P. M. VAIDYA, *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem*, Discrete Comput. Geom., 4 (1989), pp. 101–115.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 154–169, February 1994

FAT TRIANGLES DETERMINE LINEARLY MANY HOLES*

JIŘÍ MATOUŠEK†‡2, JÁNOS PACH§¶3, MICHA SHARIR§134, SHMUEL SIFRONY§, AND EMO WELZL‡24

Abstract. The authors show that for every fixed $\delta > 0$ the following holds: If F is a union of n triangles, all of whose angles are at least δ, then the complement of F has $O(n)$ connected components and the boundary of F consists of $O(n \log \log n)$ straight segments (where the constants of proportionality depend on δ). This latter complexity becomes linear if all triangles are of roughly the same size or if they are all infinite wedges.

Key words. combinatorial geometry, computational geometry, Davenport Schinzel sequences, union of geometric figures, fat triangles

AMS subject classifications. 05A99, 52A10, 52A37, 68Q20, 68R99, 68R05

1. Introduction. The problem studied in this paper is to obtain sharp upper bounds on the combinatorial complexity of the union of n geometric figures in the plane. This problem arises in many applications. For example, in motion planning for systems with two degrees of freedom, one constructs the two-dimensional configuration space of the system as the complement of the union of n "forbidden regions," each representing the space of placements of the system in which a collision occurs between two specific system and obstacle features (see [9], [14] for details). It has also been observed recently that families of figures, with the property that the union of any subfamily has small combinatorial complexity, have several additional useful properties. For example, they admit efficient output-sensitive hidden surface removal algorithms (when these figures lie at various heights and are viewed from a point far below them) [20]. Also one can obtain sharp bounds on the number of "k-sets" in an arrangement of such figures [19] and an efficient algorithm for "point-stabbing" queries in a collection of such figures (where one has to report all figures containing a query point) [19].

The simplest example of a family with the above property is a collection of half planes, each bounded by a line, or more generally by a pseudoline. A more interesting example is a family of *pseudodisks*, i.e., figures with the property that the boundaries of each pair of them intersect in at most two points. It was shown in [14] that the boundary of the union of n pseudodisks consists of at most $6n - 12$ connected pieces of the boundaries of the given figures (a special case of this result has also been obtained in [8]). Another case was studied in [4] and involved a family of figures, each bounded between a portion of the x-axis and a curve lying above the axis and delimited by two points on the axis, with the property that any pair of these curves intersect in at most 3 points. It was shown that the combinatorial complexity of the union of n such figures is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann's function.

*Received by the editors June 18, 1990; accepted for publication (in revised form) September 18, 1992. This research was partially supported by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) under National Science Foundation grant STC-88-09648.

†Department of Computer Science, Charles University, Praha, Czechoslovakia.

‡Institut für Informatik, Freie Universität Berlin, Berlin, Germany.

§Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

¶Mathematical Institute of the Hungarian Academy of Sciences, Budapest, Hungary. The research of this author was supported by Hungarian Science Foundation grant OTKA-1814.

[1] School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The research of this author was supported by Office of Naval Research grant N00014-90-J-1284, grants from the United States–Israeli Binational Science Foundation, and the Fund for Basic Research administered by the Israeli Academy of Sciences.

[2] The research of this author was supported in part by the ESPRIT II Basic Research Actions Program of the EC under contract 3075 (Project ALCOM).

[3] The research of this author was supported by National Science Foundation grant CCR-89-01484.

[4] The research of this author was supported by a grant from the German–Israeli Foundation for Scientific Research and Development.

As all these examples indicate, the property of having a union of small combinatorial complexity somehow seems to require that the boundaries of any pair of the given figures intersect in a small number (1, 2, or 3) of points. When the allowed number of intersections becomes 4 or more, there are sets of n triangles whose union has quadratic complexity. However, one observes that to attain quadratic complexity, it seems to be essential that the triangles be very narrow and many must have an angle that tends to 0 as n increases.

The purpose of this paper is to show that if this is not allowed—namely, if we are given a collection of triangles that are "fat"—then indeed the combinatorial complexity of their union is small.

Statement of results. We call a triangle T *δ-fat*, if each angle of T is at least δ. By a *figure* we mean a (closed) region in the plane, bounded by a closed Jordan curve or by an unbounded Jordan arc.

Let $\mathcal{F}$ be a finite family of figures. A *hole* of $\mathcal{F}$ is a connected component of the complement of the union of the figures of $\mathcal{F}$. The number of holes of $\mathcal{F}$ will be denoted by $\mathrm{H}(\mathcal{F})$.

A point of the boundary of the union of a family $\mathcal{F}$ is called a *corner* of $\mathcal{F}$ if it is a point of intersection between the boundaries of two figures in $\mathcal{F}$. The *boundary complexity* of $\mathcal{F}$ (denoted by $\mathrm{BC}(\mathcal{F})$) will be the number of corners of $\mathcal{F}$; note that we do not count vertices (if any) of the figures of $\mathcal{F}$ as corners — their number is usually small and presents no problems in the analysis. An *edge* of $\mathcal{F}$ is a connected portion of the boundary of the union of $\mathcal{F}$ contained in the boundary of a single figure between two adjacent corners.

Our main results are the following theorems:

THEOREM 1.1. *For any fixed $\delta > 0$, every family $\mathcal{F}$ of n δ-fat triangles has $O(n)$ holes, with the constant of proportionality depending on δ.*

Using this theorem in combination with the Combination Lemma of Edelsbrunner et al. [6] (reviewed in the next section), we will show in §4 the following

THEOREM 1.2. *For any fixed $\delta > 0$, the boundary complexity of every family $\mathcal{F}$ of n δ-fat triangles is $O(n \log\log n)$ (again, the constant of proportionality depends on δ). On the other hand, there exist such families (even with $\delta = 60°$) whose boundary complexity is $\Omega(n\alpha(n))$.*

In the special case when the triangles in our family all have roughly the same size, the boundary complexity becomes linear (in the statement of the theorem, $\mathrm{diam}(T)$ denotes the diameter of triangle T):

THEOREM 1.3. *Let $\delta > 0$ and $0 < c \leq C$ be fixed numbers. Let $\mathcal{F}$ be a family of n δ-fat triangles, such that $c \leq \mathrm{diam}(T) \leq C$ for every triangle $T \in \mathcal{F}$. Then the boundary complexity of $\mathcal{F}$ is $O(n)$ (with the constant of proportionality depending on δ and on C/c). The boundary complexity is also linear for a family of δ-fat wedges (regions bounded between a pair of rays with a common endpoint).*

Related results have been recently obtained by Alt et al. [2], where the complexity of fat objects was first considered. They showed, among other results, that the boundary complexity of the union of n δ-fat double wedges is $O(n)$. They have also shown that the number of holes (and the boundary complexity) of the union of n triangles, each of which is homothetic either to a fixed triangle T or to the reflection of T, is linear. These results are special cases of the results that we obtain in this paper.

2. Preliminaries. In this section we review two basic results concerning arrangements of certain types of figures, which will be needed in the subsequent analysis. The first result, adapted from [6], is stated here in a more specialized form, which nevertheless follows easily from the original version of [6].

LEMMA 2.1 (Combination Lemma [6]). *Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be families of figures, whose boundaries are polygons with n_1 and n_2 sides in total. Then*

$$\mathrm{BC}(\mathcal{F}_1 \cup \mathcal{F}_2) \leq \mathrm{BC}(\mathcal{F}_1) + \mathrm{BC}(\mathcal{F}_2) + O(n_1 + n_2 + \mathrm{H}(\mathcal{F}_1 \cup \mathcal{F}_2)).$$

The next lemma follows from a more general statement about pseudodisks, [14], [8]. However, for the sake of completeness we present the simple proof for the special case we need here (two figures are called *homothetic*, if one can be obtained from the other by translation and scaling).

LEMMA 2.2. *For a family $\mathcal{F}$ of n pairwise homothetic triangles we have* $\mathrm{BC}(\mathcal{F}) \leq 6n$.

Proof. Let us first observe that the boundaries of two homothetic triangles cross in at most two points. Consider now a corner w of $\mathcal{F}$, which is the intersection of two edges e and e' of two of the triangles. Each edge has one direction at the corner in which the edge 'disappears' locally into the respective other triangle. Let v and v' be the vertices incident to the edges in those distinguished directions. Note that either v or v' must be covered by the respective other triangle. Indeed, in order for v to lie outside, the edge e must create another boundary crossing, and similarly for v'; thus, if both v and v' are not covered, we get at least three boundary crossings, which is impossible. If v is covered, the corner w is the last corner on e in the direction toward v (since, by convexity, the whole portion between w and v is covered); an analogous statement holds for v'.

We charge the corner to the pair (e, v), if v lies in the other triangle, and to the pair (e', v'), otherwise. We have seen that each such pair can be charged at most once, so the number of corners is at most twice the number of vertices, namely, $6n$. (Note that this bound holds even if we also count in the boundary complexity the triangle vertices on the boundary. □

3. Bounding the number of holes. In this section we prove Theorem 1.1, that is, we show that a set of δ-fat triangles has at most a linear number of holes.

Passing to canonical triangles. The first step in the proof is to transform the given collection $\mathcal{F}$ to another collection consisting of *canonical triangles* so that the number of holes in the new collection is not much different than the number of holes of $\mathcal{F}$. Specifically, we have

LEMMA 3.1 (Canonization Lemma). *For each $\delta > 0$ there exists a positive constant $c = c(\delta) = O(1/\delta)$, such that if $\mathcal{F}$ is a family of n δ-fat triangles, then there exists families $\mathcal{F}_1, \ldots, \mathcal{F}_c$ consisting of $O(n)$ triangles in total, such that each $\mathcal{F}_i$ is a family of $\delta/4$-fat homothetic triangles and*

$$\mathrm{H}(\mathcal{F}) \leq \mathrm{H}(\mathcal{F}_1 \cup \cdots \cup \mathcal{F}_c) + O(n).$$

The canonization is achieved by producing triangles which have edges from some fixed finite set of directions $D(\delta) = \{0, \hat{\delta}, 2\hat{\delta}, \ldots, (k_\delta - 1)\hat{\delta}\}$, where $k_\delta = \lceil 4\pi/\delta \rceil$ and $\hat{\delta} = 2\pi/k_\delta$. The set $D(\delta)$ has the property that every angle of at least $\delta/2$ contains a direction in $D(\delta)$.

LEMMA 3.2. *Let $\delta > 0$. Any δ-fat triangle can be expressed as the union of three $(\delta/2)$-fat triangles T_1, T_2, T_3, such that two of the sides of each T_i have directions in $D(\delta)$, while the third is a side of T.*

Proof. Let T be a δ-fat triangle with vertices A, B, C, and let O be the center of its inscribed circle (which is also the intersection of the angle bisectors, see Fig. 1). Hence each of the angles OAC, OBC is at least $\delta/2$. We can thus find a point Q in the triangle, such that the point O lies in the triangle ABQ and the segments AQ and BQ have directions in $D(\delta)$. Such a point Q determines the triangle $T_1 = ABQ$, and T_2, T_3 can be constructed in an analogous manner for the two other sides of T. □

In the first stage of canonization, we replace each triangle in $\mathcal{F}$ by three "semi-canonical" triangles as in the preceding lemma. In a second stage we shrink each of the new triangles until it becomes the union of two "fully canonical" triangles. This is shown in the following lemma.

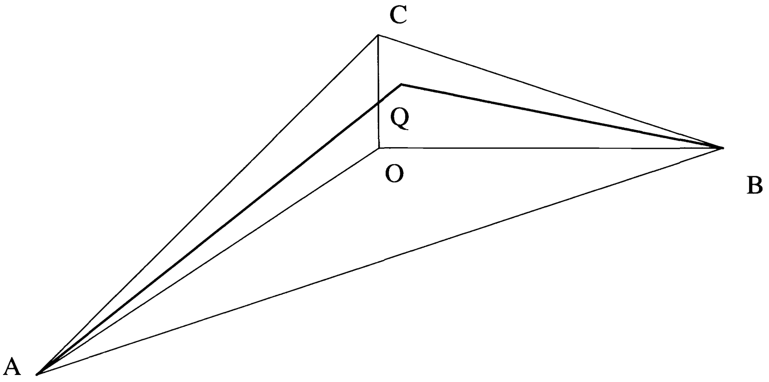


FIG. 1. *First stage of canonization.*

LEMMA 3.3 (Shrinking Lemma). *Let $\mathcal{F}$ be a family of n triangles. Let $\mathcal{F}^\star$ arise from $\mathcal{F}$ by replacing each triangle $T = ABC$ in $\mathcal{F}$ by the union of two triangles ABX, AYC, such that X lies on AC and Y lies on AB (see Fig. 2). Then*

$$\mathrm{H}(\mathcal{F}) \leq \mathrm{H}(\mathcal{F}^\star) + 3n.$$

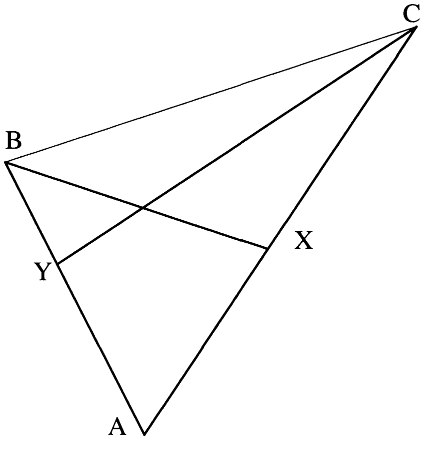


FIG. 2. *Second stage of canonization.*

Proof. Since the union of $\mathcal{F}^\star$ is contained in the union of $\mathcal{F}$, the only way in which the number of holes of $\mathcal{F}$ might decrease as we pass from $\mathcal{F}$ to $\mathcal{F}^\star$ is when a pair of holes are merged together to form a single hole. Let us imagine that every triangle ABC of $\mathcal{F}$ shrinks into the corresponding figure of $\mathcal{F}^\star$ by a continuous deformation, during which the side BC is deformed into an outward-concave curve γ, e.g., in the manner depicted in Fig. 3.

During this shrinking process, two holes of $\mathcal{F}$ may be merged to form a new hole only when a vertex of some other triangle is passed by γ and appears on the boundary of the union of the shrinking family of figures. Each such event decreases the number of holes by 1, and we can charge this event to the newly appearing vertex. Note that this event is irreversible—once a vertex has appeared on the boundary of our family, it will never be covered again, so there are at most $3n$ such events during the entire shrinking process, hence the number of holes could not have decreased by more than $3n$. □

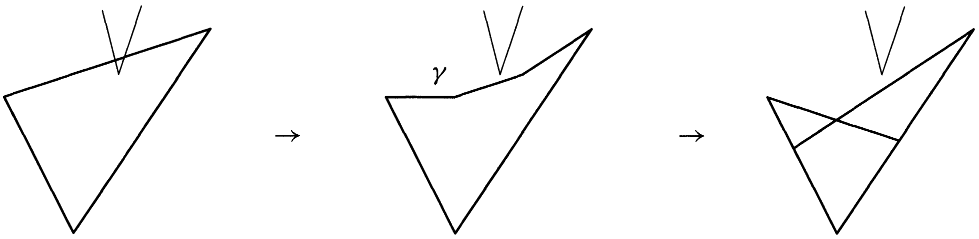


FIG. 3. *Shrinking a triangle.*

Now the proof of Lemma 3.1 is easy. First we replace, using Lemma 3.2, each triangle of the original family $\mathcal{F}$ by the union of a triple of semicanonical triangles, each having two sides in the set of canonical directions. Then we replace each semicanonical triangle ABC by a pair of triangles ABX, AYC as in Lemma 3.3 so that each side of the new triangles has a direction in a fixed finite set of directions and one angle in each triangle is exactly $\hat{\delta}$ (the angle at vertex B and C, respectively); thus the final triangles fall already into a constant number of families of homothetic triangles). We can apply the shrinking of Lemma 3.3 once more to ensure that we have a set of at most $12n$ triangles, where two angles are $\hat{\delta}$. That is, the triangles fall now in $2k_{\delta} = O(1/\delta)$ homothetic classes. Lemma 3.3 is easily seen to imply that at most $O(n)$ holes can be lost in both shrinking processes, since the number of triangles (and so the number of vertices) is linear. □

Boundary complexity for a pair of homothetic families. By the Canonization Lemma 3.1, it suffices to bound the number of holes of a union of a constant number of families, each consisting of homothetic $\hat{\delta}$-fat triangles. For simplicity of exposition, we will continue to denote $\hat{\delta}$ by δ. If $\mathcal{F} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_c$, then any corner of $\mathcal{F}$ must be a corner of some family of the form $\mathcal{F}_i \cup \mathcal{F}_j$, for $1 \leq i, j \leq c$ (this also includes corners that arise within a single family $\mathcal{F}_i$), thus

$$\mathrm{BC}(\mathcal{F}) \leq \sum_{i,j} \mathrm{BC}(\mathcal{F}_i \cup \mathcal{F}_j).$$

Therefore, Theorem 1.1 will be proved if we prove the following:

LEMMA 3.4. *Let* $\delta > 0$ *be fixed. Let* $\mathcal{F}_1$ *and* $\mathcal{F}_2$ *be families of triangles, each consisting of* n δ*-fat homothetic triangles. Then* $\mathrm{BC}(\mathcal{F}_1 \cup \mathcal{F}_2) = O(n)$ *(with a constant of proportionality that depends on* δ*).*

Proof. Let us put $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$. We will bound the number of edges of the union F of $\mathcal{F}$. Let us call the edges of the union F_i of $\mathcal{F}_i$ the *superedges* of $\mathcal{F}_i$, $i = 1, 2$. If e is an edge of F lying on a superedge s of a triangle T of $\mathcal{F}_i$, we call s the *supporting superedge*, T the *supporting triangle*, and $\mathcal{F}_i$ the *supporting family* of e.

By Lemma 2.2, we know that the boundary complexity of $\mathcal{F}_1$ and of $\mathcal{F}_2$ are linear; i.e., the number of superedges is linear in n.

Call an edge e of F *trivial* if e is the first or the last edge of F along its supporting superedge. The number of trivial edges is therefore $O(n)$.

Since edges of F have only six possible directions, it suffices to bound the number of nontrivial edges with one fixed direction. Fix such a direction d, and let e be a nontrivial edge of F having direction d. Suppose e is supported by the family $\mathcal{F}_2$. The edge e is adjacent to two edges f and f', whose respective supporting triangles T and T' belong to the other family $\mathcal{F}_1$ and are thus homothetic. Let s and s' be the supporting superedges of f and f', respectively.

Call the pair (s, s') an *active pair* of superedges, if they are connected by an edge e as above; we will refer to e as an edge *belonging* to (s, s').

We claim that the number of active pairs is $O(n)$. Indeed, the superedges of $\mathcal{F}_1$ are nonintersecting and each active pair is visible from each other in direction d. The number of such visible pairs is linear; this can be seen by sweeping a line in direction d across the plane or by applying a graph planarity argument.

The proof will therefore be finished if we prove the following:

LEMMA 3.5. *Let (s, s') be an active pair of superedges; then the number of nontrivial edges belonging to (s, s') is bounded by a constant (depending on δ).*

Proof. Let T and T' be the triangles supporting s and s', respectively (see Fig. 4). Consider all the edges belonging to the active pair (s, s'), which, by our convention, are all assumed to have direction d; without loss of generality we assume that d is horizontal and that T lies to the left of T' (see Fig. 4). Without loss of generality, we may also assume that the corresponding holes of $\mathcal{F}$ lie below these edges.

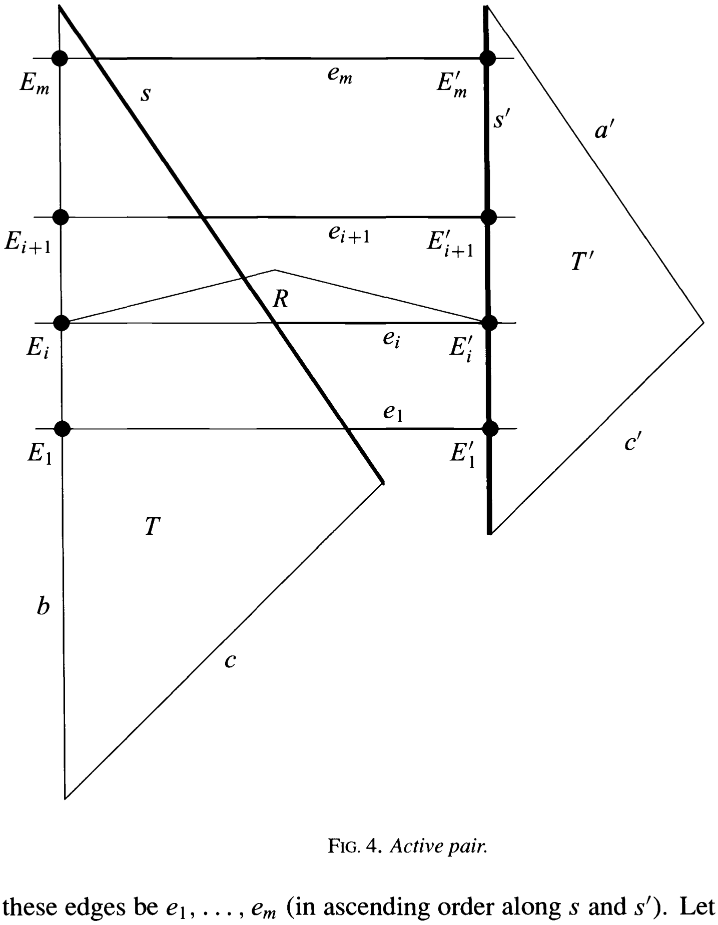


FIG. 4. *Active pair.*

Let these edges be $e_1, \ldots, e_m$ (in ascending order along s and s'). Let b denote the side of T parallel to s', and let a' denote the side of T' parallel to s. Without loss of generality we may assume that the projection of s in direction d on the line containing b is contained in b (and similarly for a' and s').

For each edge e_i let us denote the intersections of its superedge with the edges b, s' by E_i, E'_i, respectively (see Fig. 4; these intersections exist by the assumption just made and because each e_i is nontrivial and thus penetrates through both T and T'). Consider the parallelogram

$E_1 E_m E'_m E'_1$. The angle $E_1 E_m E'_1$ is at least δ; hence, $|E_1 E'_1| \geq \gamma \cdot |E_1 E_m|$, where $\gamma > 0$ is a constant depending on δ.

Consider an edge e_i and its supporting triangle T_i. This triangle must contain both points E_i and E'_i. The key observation is that T_i cannot intersect the segment $E_{i+1} E'_{i+1}$, simply because $E_{i+1} E'_{i+1}$ is part of a superedge and T_i belongs to the same family as $E_{i+1} E'_{i+1}$.

Since T_i is δ-fat, it must contain the triangle R with base $E_i E'_i$ and angles δ at the vertices E_i and E'_i, so R also does not intersect $E_{i+1} E'_{i+1}$. This means that the length of the segment $E_i E_{i+1}$ is at least a constant fraction (depending on δ) of the length of $E_i E'_i$ and, hence, also of $E_1 E_m$. This implies that the number m of nontrivial edges belonging to the active pair (s, s') is bounded by a constant. □

Remark. A more detailed analysis in the previous lemma shows that the constant claimed is $O(1/\delta^2)$. That is, if we denote the cardinality of $\mathcal{F}_i$ by n_i, then $\mathrm{BC}(\mathcal{F}_1 \cup \mathcal{F}_2) = O(n_i + n_j + \min\{n_i, n_j\}/\delta^2)$. This gives a bound of $O(cn/\delta^2) = O(n/\delta^3)$ for $\mathrm{H}(\mathcal{F})$ (for the original family $\mathcal{F}$). Summing up, we have at most $O(n/\delta^3)$ holes in the union of n δ-fat triangles. This is probably not tight in terms of δ; the best lower bound we can derive is $\Omega(n/\delta)$.

In closing this section, we note that Lemma 3.4 has the following corollary, which may be of independent interest. Call a family of triangles *c-oriented* if the orientations of the edges of the triangles are drawn from a fixed set of c orientations; see [10], [11], [18], [22] for several studies of c-oriented polygons.

COROLLARY 3.6. *The boundary complexity of a family of n c-oriented triangles is $O(n)$, where the constant of proportionality depends on c and the minimum angle between any two of the c given orientations.*

Remark. The weaker result of Alt et al. [2] is also a special case of this corollary.

4. The boundary complexity of the union of fat triangles. In this section we analyze the boundary complexity of the union of n fat triangles. We rely on the results of the preceding section concerning the number of holes, on the Combination Lemma 2.1, and on a special way of decomposing the given collection of triangles into subcollections, each having a union with small boundary complexity.

Proof of Theorem 1.2. Let $\mathrm{BC}(n)$ denote the maximum possible boundary complexity of a family of n δ-fat triangles. Let $\mathcal{F}$ be such a family. Applying the first canonization step in the proof of Theorem 1.1, we replace $\mathcal{F}$ by a constant number of subfamilies, each consisting of triangles that have two sides with fixed orientations. By further refining this partitioning, we can also assume that within each subfamily the orientations of the third edges of the triangles all lie within some small angular interval, of length, say 1°. Note that the number of subfamilies is still a constant and that the overall union of all subfamilies is equal to the union of $\mathcal{F}$. We will show that the boundary complexity of the union of the triangles in the ith subfamily is $O(n_i \log\log n_i)$, where n_i is the number of triangles in the subfamily. The Combination Lemma 2.1 then implies that the boundary complexity of the union of all triangles is $O(n \log\log n)$, as asserted.

Thus, from now on, we consider a single subfamily, which, for simplicity, we also denote by $\mathcal{F}$. By applying an appropriate affine transformation, we can assume that each triangle is a right triangle with one horizontal edge and one vertical edge, that these edges meet in the lower-left vertex of the triangle, and that the hypotenus of the triangle has orientation between, say, 134 and 136 degrees (so the triangle is nearly isosceles).

Our first step is to partition $\mathcal{F}$ into $O(\log n)$ subfamilies so that the boundary complexity of each subfamily is almost linear in the number of triangles it contains.

LEMMA 4.1. *If all triangles in $\mathcal{F}$ have the form assumed above and meet a common horizontal line, then* $\mathrm{BC}(\mathcal{F}) = O(n \cdot 2^{\alpha(n)})$, *where $\alpha(n)$ is the inverse Ackermann function.*

Proof. Without loss of generality, assume the line is the x-axis. For each triangle $T \in \mathcal{F}$ let T^+ denote its portion above the x-axis and T^- denote its portion below the x-axis. The boundary complexity of $\mathcal{F}$ is clearly bounded by the sum of the boundary complexities of the union of the triangles T^+ and of the union of the trapezoids T^-. The boundary complexity of the upper triangles T^+ is $O(n)$—if we direct all edges toward the x-axis, then, as is easily seen, every corner is the last corner (in this direction) for one of its two edges.

As to the lower trapezoids T^-, we first decompose each T^- into two interior-disjoint portions, one being an axis-parallel rectangle and the other being a right, nearly isosceles triangle with a horizontal edge and a vertical edge whose top vertex lies on the x-axis; see Fig. 5.

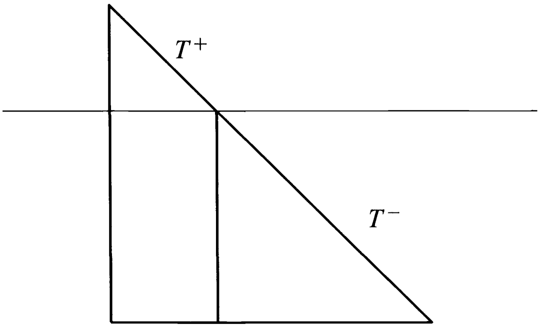


FIG. 5. *A triangle cut by a horizontal line.*

It suffices to show that the boundary complexity of the union of the family $\mathcal{F}^-$ consisting of these new triangles is $O(n \cdot 2^{\alpha(n)})$, because the boundary complexity of the union of the rectangles is trivially linear and the Combination Lemma 2.1 implies that merging the rectangles with the triangles of $\mathcal{F}^-$ yields a joint boundary complexity that is proportional to the complexity of $\mathcal{F}^-$. We therefore restrict our attention only to the union of $\mathcal{F}^-$.

CLAIM 1. A horizontal edge of a triangle T in $\mathcal{F}^-$ can be incident to at most four hole corners.

Proof. Let e be the given edge, and let $e' = XY$ be an interval along e that appears on the boundary of the union and is not the leftmost such interval along e. The left endpoint X of e' is the intersection of e with the hypotenus of another triangle T', and our assumption concerning e' implies that the vertical edge of T' also cuts e. See Fig. 6. Let the top angles of T, T' be α, α' respectively, and let the length of the vertical edge of T be h; let g denote the intersection $e \cap T'$.

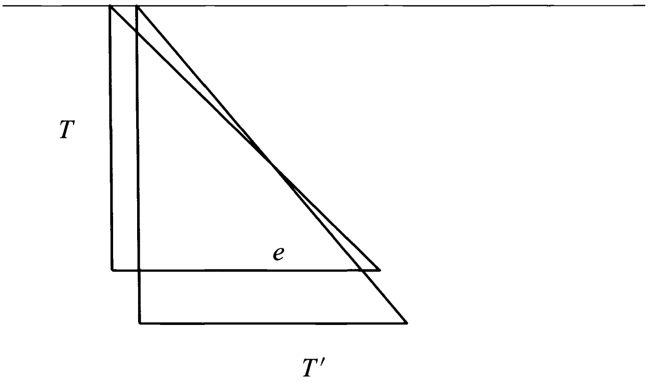


FIG. 6. *Two "interleaving" triangles.*

We have $|e| = h \tan\alpha$ and $|g| = h \tan\alpha'$. Thus $|g|/|e| = \tan\alpha'/\tan\alpha$ is very close to 1; in particular, it is greater than 1/2. This shows that the interval e' is unique and so that e can contain at most two intervals that bound holes, namely, e' and another leftmost interval. This completes the proof of the claim. □

CLAIM 2. The total number of hole corners that are incident to either a horizontal edge or to a vertical edge is $O(n)$.

Proof. Claim 1 implies that the number of hole corners along horizontal edges is $O(n)$. Let e be a vertical edge, and let e' be an interval along e bounding a hole. It is easily verified that the top endpoint of e' must be incident to a horizontal edge. The claim is now immediate. □

It therefore remains to consider only hole corners formed by intersections of two hypotenuses of the triangles of $\mathcal{F}^-$. We order these corners in lexicographical order so that $c_1 \prec c_2$ if, for $c_1 = (x_1, y_1)$ and $c_2 = (x_2, y_2)$, either $x_1 < x_2$ or $x_1 = x_2$ and $y_1 < y_2$. This is clearly a linear order.

Our strategy is to transform this sequence of corners to a *Davenport Schinzel sequence* of order 4 [1], [12], which will then yield the asserted bound on the boundary complexity of $\mathcal{F}^-$. (Recall that a Davenport Schinzel sequence of order 4 is a sequence that does not have any two equal adjacent elements and does not contain as a (not necessarily contiguous) subsequence an alternation $a \cdots b \cdots a \cdots b \cdots a \cdots b$ of length 6 between any two distinct symbols a and b.) To this end, we divide each hypotenus at its midpoint into two subsegments of equal length, which we refer to as its top part and bottom part, respectively. For each corner c consider the hypotenus incident to c and appearing along the hole just below c; c is *associated* with the part (top or bottom) of that hypotenus, to which it is incident. See Fig. 7.

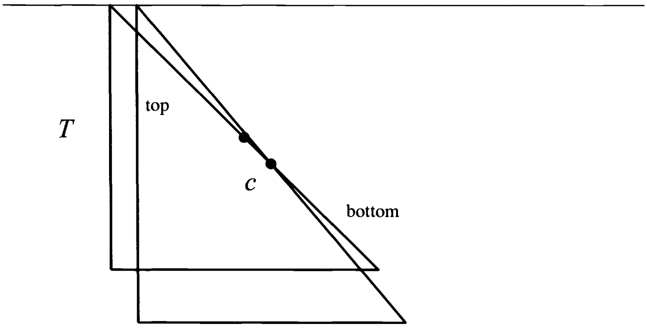


FIG. 7. *The corner c is associated with the bottom part of the triangle T.*

We proceed through the ordered sequence of corners and form a sequence U, consisting of all associated appearances of the top or bottom hypotenus parts in the order that the corresponding corners are encountered. Thus U is composed of at most $2n$ distinct symbols.

CLAIM 3. The number of appearances of bottom parts in U is at most n, and the number of pairs of equal consecutive elements in U is $O(n)$.

Proof. We first show that no bottom part of a hypotenus can appear twice in U. Indeed, let T be a triangle with a hypotenus h and c be a hole corner of the kind we consider that is associated with the bottom part of h. Thus there exists another triangle T' whose hypotenus meets T at c and has a smaller slope than h. A calculation similar to that in the proof of Claim 1 shows that the next higher appearance of h along a hole must already appear on its top part. This establishes the first assertion of the claim.

Next consider adjacent equal elements of U. Suppose a hypotenus h of some triangle T appears twice consecutively in U. Thus h contains two subintervals e, e' that bound holes.

But then the bottom endpoint c of the higher of these two intervals must be incident to a vertical edge (otherwise c is incident to some other hypotenus h', which necessarily appears in U between the two appearances of h). The claim is thus an immediate consequence of Claim 2. □

We can therefore delete from U all bottom appearances and then delete one of each pair of equal consecutive elements. The new sequence $U^\star$ consists of only top hypotenus parts (so it is composed of at most n distinct symbols), has no pair of equal adjacent elements, and satisfies $|U| = |U^\star| + O(n)$.

We claim that $U^\star$ is indeed a Davenport Schinzel sequence of order 4. That is, we have to show that $U^\star$ cannot contain an alternating subsequence of the form $a \cdots b \cdots a \cdots b \cdots a \cdots b$, where a and b are top parts of the hypotenuses of two distinct respective triangles, T, R.

Suppose to the contrary that such an alternation exists. We distinguish between two cases:

CASE I. *T and R intersect in at most two points.* This can happen in one of the four schematic forms shown in Fig. 8.

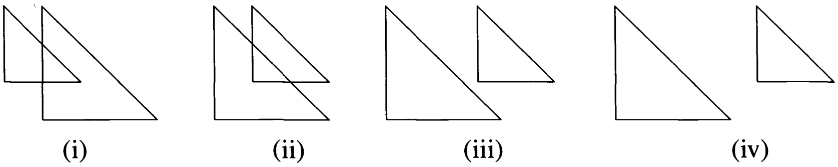


FIG. 8. *Two triangles of $\mathcal{F}^-$ intersecting in at most 2 points.*

Cases (i) and (iv) are easy, because they allow no alternation of a and b in U, as is easily checked. In case (ii) let us first assume that a is the top part of the hypotenus of the left triangle. Note that all appearances of a between the first and last appearances of b correspond to corners that lie in the vertical strip spanned by the right triangle R. Let p and q be two subintervals of a that give rise to two such appearances of a. Then it is easily seen that there must exist another triangle Q that cuts the hypotenus of T in some interval between p and q; see Fig. 9. Denote the top angles of triangles T, R, Q by $\alpha, \alpha', \alpha''$, respectively. Let d' denote the length of the vertical edge of R, let d_0 denote the vertical distance between the bottom endpoint of p and the top endpoint of q, and let d denote the vertical distance from the top endpoint of q to the x-axis; see Fig. 9.

Simple trigonometric calculations show that

$$d \tan \alpha'' = d_0 \tan \alpha < d' \tan \alpha'$$

and

$$d_0 + d' < d,$$

which is clearly impossible, since all three angles $\alpha, \alpha', \alpha''$ are close to 45°.

This argument implies that, between the first and last appearance in $U^\star$ of the hypotenus of the right triangle, there can be at most one appearance of the hypotenus of the left triangle. Thus the maximum length of an alternation between a and b in $U^\star$ is 5 (in the form $a \cdots b \cdots a \cdots b \cdots a$). If a is the top part of the hypotenus of the right triangle, the above analysis shows that the longest possible alternation is now only $a \cdots b \cdots a \cdots b$. Exactly the same analysis applies in case (iii).

CASE II. *T and R intersect in four points.* This is depicted in Fig. 10. Again without loss of generality we can assume that T is the triangle whose top vertex lies to the left of that of R (otherwise, as above, the maximum possible alternation will be shorter).

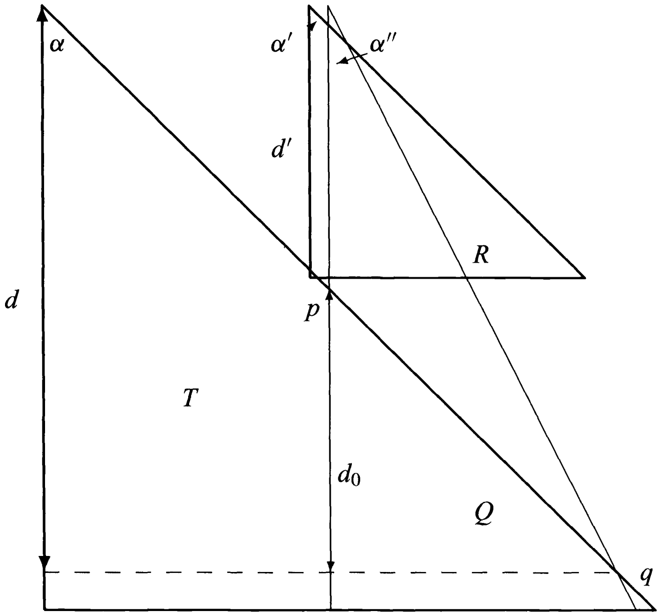


FIG. 9. *Case* I(ii) *of the proof.*

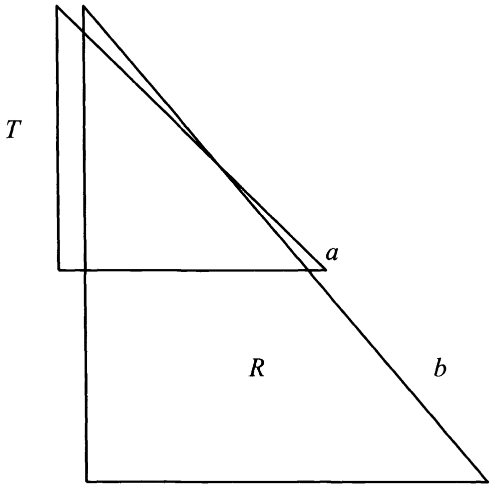


FIG. 10. *Case* II *of the proof.*

Note that the second appearance of a in the alternation must be to the right of the intersection point of the two hypotenuses, which implies that the two last appearances of b in the alternation must occur below the horizontal edge of T. But then, arguing as in the proof of Claim 1, it is easy to show that the last occurrence of a in the alternation must be at the bottom part of the hypotenus, contrary to assumption. Thus the alternation is impossible.

Hence $U^\star$ is indeed a Davenport Schinzel sequence of order 4 composed of at most n distinct symbols, so its length is at most $O(n \cdot 2^{\alpha(n)})$ [1]. This is also an upper bound on the length of U, and this clearly completes the proof of the lemma. □

We now decompose $\mathcal{F}$ as follows. We first find a horizontal line ℓ with the property that the number of triangles in $\mathcal{F}$ lying completely above ℓ and the number of triangles lying completely below ℓ are both at most $n/2$. Let $\mathcal{F}_1$ denote the subfamily of triangles of $\mathcal{F}$ intersecting ℓ. We apply the same procedure to the two subfamilies of $\mathcal{F}$ consisting respectively of the triangles lying above ℓ and of those lying below ℓ. For each of these subfamilies we find a "halving" horizontal line as above and define $\mathcal{F}_2$ to be the collection of triangles in these subfamilies which intersect one of these halving lines. We are now left with four subfamilies, each of which is next halved by a line, and $\mathcal{F}_3$ consists of the remaining triangles that intersect one of these lines. Continuing in this fashion, we obtain a decomposition of $\mathcal{F}$ into $O(\log n)$ subfamilies, $\mathcal{F}_1, \mathcal{F}_2, \ldots$, and the preceding lemma is easily seen to imply that the boundary complexity of each subfamily $\mathcal{F}_i$ is $O(n_i \cdot 2^{\alpha(n_i)})$, where $n_i = |\mathcal{F}_i|$.

We now apply the Combination Lemma 2.1 in a tree-like fashion. That is, we merge the subfamilies $\mathcal{F}_i$ two at a time, then merge each of the resulting collections two at a time, and so on, until all subfamilies are merged together. At each step, when merging two subfamilies $\mathcal{G}_1, \mathcal{G}_2$ to form a combined subfamily $\mathcal{G}$, we have

$$\mathrm{BC}(\mathcal{G}) \leq \mathrm{BC}(\mathcal{G}_1) + \mathrm{BC}(\mathcal{G}_2) + O(n_1 + n_2) ,$$

where n_i is the size of $\mathcal{G}_i$, for $i = 1, 2$. This is an immediate consequence of the Combination Lemma and of the fact that the number of holes of $\mathcal{G}$ is $O(n_1 + n_2)$. Since the depth of the tree representing these merges is $O(\log \log n)$ and the sum of the boundary complexities of the individual subfamilies $\mathcal{F}_i$ is $O(n \cdot 2^{\alpha(n)})$, it follows easily that $\mathrm{BC}(\mathcal{F}) = O(n \log \log n)$.

To obtain the lower bound in Theorem 1.2, take a collection of n line segments whose lower envelope consists of $\Omega(n\alpha(n))$ subsegments [23]. Flatten the collection in the y-direction until all segments have almost horizontal slope. Then replace each segment e by an equilateral triangle lying above e and having e as one of its sides. It is easily checked that the boundary complexity of the union of these triangles is $\Omega(n\alpha(n))$. □

Remark. By modifying the above lower bound construction and exploiting the special structure of the construction in [23], one can also obtain a collection of n equilateral triangles, whose union has $\Theta(n)$ holes, so that no triangle appears more than once along the boundary of any single hole, and yet the overall boundary complexity is $\Omega(n\alpha(n))$.

Proof of Theorem 1.3. Recall that the theorem asserts that if $\mathcal{F}$ is a family of n δ-fat triangles with $c \leq \mathrm{diam}(T) \leq C$ for all $T \in \mathcal{F}$, then $\mathrm{BC}(\mathcal{F}) = O(n)$, with the constant of proportionality depending on δ and C/c. Let $\mathcal{F}$ be such a family of triangles. We choose a real number D that satisfies the following two conditions:

(i) No square with side D is intersected by more than two sides of any triangle from $\mathcal{F}$.

(ii) The diameter of any triangle of $\mathcal{F}$ is at most a constant multiple of D.

The existence of such a D is guaranteed by the assumptions on $\mathcal{F}$; the constant factor in condition (ii) is easily seen to be of the form $\frac{C}{c}\phi(\delta)$, for an appropriate function ϕ.

Let us cover the plane by a grid of squares with side length D. By the choice of D, every triangle of $\mathcal{F}$ intersects at most a constant number of squares of this grid. We claim that the boundary complexity of $\mathcal{F}$ inside each grid square is linear in the number of triangles intersecting that square, and this will imply that the total boundary complexity of $\mathcal{F}$ is $O(n)$.

Let us consider a fixed square Q of the grid. For each triangle $T \in \mathcal{F}$, at most two sides of T intersect Q; hence, there exists a wedge (angle) W_T such that $Q \cap W_T = Q \cap T$. Let us consider the family

$$\mathcal{W} = \{W_T \ : \ T \in \mathcal{F},\ T \cap Q \neq \emptyset\}.$$

The boundary complexity of $\mathcal{W}$ is an upper bound for the complexity of the part of the boundary of $\mathcal{F}$ inside Q. Adapting Theorem 1.1 to the special case of wedges, it is easily seen that the family $\mathcal{W}$ has a linear number of holes. We claim that $\mathcal{W}$ can be partitioned into a constant number of subfamilies $\mathcal{W}_1, \ldots, \mathcal{W}_c$, each of which has a linear boundary complexity. Applying the Combination Lemma 2.1 (as in the preceding proof) a constant number of times, we obtain a linear bound on the boundary complexity of $\mathcal{W}$. (This part of the proof also establishes the second assertion in Theorem 1.3 concerning the complexity of the union of fat wedges.)

We may assume that the apex angle of each wedge of $\mathcal{W}$ is at least δ (this is obvious for triangles having two sides intersecting Q; for triangles with only one intersecting side, the choice of the apex and its angle are fairly arbitrary). It follows that there exists a fixed set of a constant number $c = O(1/\delta)$ of canonical orientations (e.g., $\delta/2$ apart from each other) so that each wedge in $\mathcal{W}$ contains a ray emerging from its apex and having one of these canonical orientations. We thus choose the decomposition $\mathcal{W} = \mathcal{W}_1 \cup \cdots \cup \mathcal{W}_c$ so that for all wedges in the same subfamily, the corresponding rays are all in the same (canonical) direction. It is well known that the boundary complexity of each subfamily $\mathcal{W}_i$ of wedges is linear. Indeed, if the common ray direction is assumed to be the negative y-direction, the boundary of the union of $\mathcal{W}_i$ is the upper envelope of the collection of rays that bound these wedges, and it is known that the complexity of such an envelope is linear (see, e.g., [5]). This finishes the proof of Theorem 1.3. □

5. Extensions, applications, and open problems. We have so far shown that the union of n δ-fat triangles has a linear number of holes and that its boundary complexity is $O(n \log\log n)$ and can be $\Omega(n\alpha(n))$. In this section we consider several extensions of these results, mention some applications, and conclude with some open problems.

Constructing the union of fat triangles. First we note that one can also calculate efficiently the union of such a family $\mathcal{F}$. The following algorithm, adapted from [14], can be used. Partition $\mathcal{F}$ into two subfamilies of roughly $n/2$ triangles each. Recursively calculate the union F_1 of $\mathcal{F}_1$ and the union F_2 of $\mathcal{F}_2$. Then merge the two unions by the line-sweeping procedure of Chazelle and Edelsbrunner [3] or of Mairson and Stolfi [15]. This computes all k intersections between the boundaries of F_1 and of F_2 in time $O(N \log N + k)$, where N is the overall size of F_1 and of F_2. But each such intersection is easily seen to be a corner of the overall union of $\mathcal{F}$, so by Theorem 1.2 we have that both N and k are bounded by $O(n \log\log n)$. This easily implies that we can construct the union of $\mathcal{F}$ from F_1 and F_2 in time $O(n \log n \log\log n)$, so the overall running time of this algorithm is $O(n \log^2 n \log\log n)$. We thus have

THEOREM 5.1. *One can calculate the union of n δ-fat triangles in $O(n \log^2 n \log\log n)$ time and $O(n \log\log n)$ storage (where the constant of proportionality depends on δ).*

Remark. One should contrast the problem of explicit construction of the union of a collection of figures to that of computing various measures of the union, such as its area or the length of its boundary. Such measures can be calculated efficiently for the case of axis-parallel rectangles, not necessarily δ-fat [17]. However, such efficient procedures are not known for general non-δ-fat collections. For δ-fat collections they are immediate by-products of the algorithm given above.

Recently, after the original submission of the paper, Miller and Sharir [16] obtained an improved randomized incremental algorithm for computing the union of n fat triangles, using $O(n \cdot 2^{\alpha(n)} \log n)$ expected time and storage.

General "fat" objects. We can also extend our results to families of polygons, which can be expressed as the union of a constant number of δ-fat triangles. Some "fatness" condition is clearly essential for such a result to hold, since one can form a quadratic number of holes with very narrow objects. Moreover, the following example shows that even when the polygons appear to be fat in an intuitive sense, they can still form quadratically many holes, so a stronger condition, such as imposed above, has to be enforced.

EXAMPLE 5.2. There exists a family of n similar convex figures (actually regular polygons), for each of which the ratio between the radii of the circumscribed and inscribed circles is less than 2 and which determine $\Omega(n^2)$ holes.

Proof. We will construct a family of $2n$ regular n-gons. Let us choose a regular n-gon $A = A_1A_2 \cdots A_n$. On each of its sides, A_iA_{i+1}, choose $n+1$ equidistant points $B_{i,0} = A_i, B_{i,1}, \ldots, B_{i,n-1}, B_{i,n} = A_{i+1}$. The first half of our family consists of n regular n-gons $C_1, C_2, \ldots, C_n$, where $C_j = B_{1,j}B_{2,j} \cdots B_{n,j}$. The second half of the family consists of n regular n-gons $D_1, \ldots, D_n$, where D_i arises as a mirror image of A, reflected around the side A_iA_{i+1}. This family determines quadratically many holes. □

Remarks. (1) This example is somewhat misleading because we ignore here the overall description complexity of the polygons C_i, D_i (which is itself quadratic). We include this example only to demonstrate that one needs to be careful in the definition of fatness if one wishes to extend the results of this paper to more complex figures than triangles.

(2) The reason for the large complexity in this example is that the boundaries of the convex figures intersect in many points per pair. It remains to investigate what happens if we consider a family of fat objects, such that the number of intersections of boundaries of any pair is bounded by a constant.

Applications. As briefly mentioned in the introduction, the fact that the boundary complexity of a family of fat triangles is small has various combinatorial and algorithmic consequences. So far these applications were limited to the case of pseudodisks and to a few other favorable cases mentioned in the introduction. These applications can now be extended to the case of fat triangles. We list some of them as corollaries of the bounds obtained in the preceding sections and omit the proofs, which are easily obtained by adapting the earlier proofs cited below.

COROLLARY 5.3. *Let $T_1, \ldots, T_n$ be n δ-fat triangles lying in three-dimensional space in arbitrary horizontal planes and viewed from a point at $z = -\infty$. Then one can perform hidden surface removal for this scene in time $O(n^{3/2} \log n (\log\log n)^{1/2} + k)$, where k is the size of the resulting "visibility map."*

Proof. See [20]. □

Remark. Recently, after the original submission of this paper, this result has been significantly improved in [13]. The algorithm presented there is also based on the results of this paper, and its running time is $O((n \log\log n + k)\log^2 n)$.

COROLLARY 5.4. *Let $T_1, \ldots, T_n$ be n δ-fat triangles in the plane, and let $k \leq n-2$ be an integer. The number of intersection points of the boundaries of these triangles which are covered by at most k other triangles is $O(nk \log\log \frac{n}{k})$.*

Proof. See [19]. □

COROLLARY 5.5. *Let $T_1, \ldots, T_n$ be n δ-fat triangles in the plane. One can preprocess them by a randomized algorithm, whose expected running time is $O(n \log^2 n \log\log n)$, into a data structure of size $O(n \log n \log\log n)$, so that, given any query point z, all k triangles containing z can be reported in (worst-case) time $O((k+1)\log n)$.*

Proof. See [19]. □

Remark. The bounds stated in the preceding theorems follow from the bound $O(n \log\log n)$ on the boundary complexity of a collection of fat triangles. Since we believe that this bound is not tight (see below), we expect corresponding improvements in the bounds of the preceding theorems. We also note that the running time of the algorithm of Corollary 5.5 can be slightly improved by the recent technique of [16] mentioned above.

Recently, the results of this paper have been applied in [21] to obtain efficient algotihms for motion planning among fat obstacles.

Open problems. The main open problem that arises is to close the gap between the upper and lower bounds on the maximum boundary complexity of a union of n δ-fat triangles. We venture the conjecture that the correct bound is indeed $O(n\alpha(n))$. It is annoying that we were unable to prove this even in the special case of Lemma 4.1.

Another problem is to calibrate the dependence of the constants of proportionality in the various bounds obtained above in terms of δ so that sharp bounds can be obtained also in cases where δ does depend on n. Some progress toward this goal was recently achieved in [7].

Finally, it is challenging to extend the results of this paper to three dimensions. For example, can one show that the boundary complexity of the union of n arbitrary cubes (or of 'fat' simplices) in 3-space is close to *quadratic* in n (as opposed to a trivial cubic upper bound)?

REFERENCES

[1] P. Agarwal, M. Sharir, and P. Shor, *Sharp upper and lower bounds on the length of general Davenport Schinzel sequences*, J. Combin. Theory Ser. A, 52 (1989), pp. 228–274.

[2] H. Alt, R. Fleischer, M. Kaufmann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig, *Approximate motion planning and the complexity of the boundary of the union of simple geometric figures*, in Proc. 6th Symposium on Computational Geometry, Berkeley, CA, 1990, pp. 281–289.

[3] B. Chazelle and H. Edelsbrunner, *An optimal algorithm for intersecting line segments in the plane*, J. Assoc. Comput. Mach., 39 (1992), pp. 1–54.

[4] H. Edelsbrunner, L. Guibas, J. Hershberger, J. Pach, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink, *On arrangements of Jordan arcs with three intersections per pair*, Discrete Comput. Geom., 4 (1989), pp. 523–539.

[5] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir, *Arrangements of curves in the plane—topology, combinatorics, and algorithms*, Theoret. Comput. Sci., 92 (1992), pp. 319–336.

[6] H. Edelsbrunner, L. Guibas, and M. Sharir, *The complexity and construction of many faces in arrangements of lines and of segments*, Discrete Comput. Geom., 5 (1990), pp. 161–196.

[7] A. Efrat, G. Rote, and M. Sharir, *On the union of fat wedges and separating a collection of segments by a line*, to appear in Computational Geometry—Theory and Applications.

[8] S. Fortune, *A fast algorithm for polygon containment by translation*, in Proc. 12th Internat. Colloq. on Automata, Languages and Programming, 1985, pp. 189–198.

[9] L. Guibas, M. Sharir, and S. Sifrony, *On the general motion planning problem with two degrees of freedom*, Discrete Comput. Geom., 4 (1989), pp. 491–521.

[10] R. H. Gütting, *Conquering contours: efficient algorithms for computational geometry*, Ph.D. Dissertation, Dortmund University, Dortmund, Germany, 1983.

[11] R. H. Gütting and Th. Ottmann, *New algorithms for special cases of hidden line elimination problems*, Computer Vision, Graphics, and Image Processing, 40 (1987), pp. 188–204.

[12] S. Hart and M. Sharir, *Nonlinearity of Davenport Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151–177.

[13] M. Katz, M. Overmars, and M. Sharir, *Efficient output sensitive hidden surface removal for objects with small union size*, Computational Geometry—Theory and Applications, 2 (1992), pp. 223–234.

[14] K. Kedem, R. Livne, J. Pach, and M. Sharir, *On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles*, Discrete Comput. Geom., 1 (1986), pp. 59–71.

[15] H. Mairson and J. Stolfi, *Reporting and counting intersections between two sets of line segments*, in Theoretical Foundations of Computer Graphics and CAD, R. A. Earnshaw, ed., NATO ASI Series, Vol. F-40, Springer-Verlag, Berlin, 1988, pp. 307–325.

[16] N. Miller and M. Sharir, *Efficient randomized algorithms for constructing the union of fat triangles and of pseudodiscs*, manuscript, 1991.

[17] F. Preparata and M. Shamos, *Computational Geometry—An Introduction*, Springer-Verlag, Berlin, 1985.

[18] G. Rawlins and D. Wood, *Optimal computation of finitely oriented convex hulls*, Inform. and Comput., 72 (1987), pp. 150–166.

[19] M. Sharir, *On k-sets in arrangements of curves and surfaces*, Discrete Comput. Geom., 6 (1991), pp. 593–613.

[20] M. SHARIR AND M. OVERMARS, *A simple output sensitive hidden surface removal algorithm*, ACM Trans. Graphics, 11 (1992), pp. 1–11.

[21] A. F. VAN DER STAPPEN, D. HALPERIN, AND M. OVERMARS, *The complexity of free space for a robot moving amidst fat obstacles*, manuscript, 1992.

[22] X. H. TAN, T. HIRATA, AND Y. INAGAKI, *The intersection searching problem for c-oriented polygons*, Inform. Process. Lett., 37 (1991), pp. 201–204.

[23] A. WIERNIK AND M. SHARIR, *Planar realization of nonlinear Davenport Schinzel sequences by segments*, Discrete Comput. Geom., 3 (1988), pp. 15–47.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 170–184, February 1994

012

THREE-DIMENSIONAL STATISTICAL DATA SECURITY PROBLEMS[*]

ROBERT W. IRVING[†] AND MARK R. JERRUM[‡]

Abstract. Suppose there is a three-dimensional table of cross-tabulated nonnegative integer statistics, and suppose that all of the row, column, and "file" sums are revealed together with the values in some of the individual cells in the table. The question arises as to whether, as a consequence, the values contained in some of the other (suppressed) cells can be deduced from the information revealed.

The corresponding problem in two dimensions has been comprehensively studied by Gusfield [*SIAM J. Comput.*, 17 (1988), pp. 552–571], who derived elegant polynomial-time algorithms for the identification of any such "compromised" cells, and for calculating the tightest bounds on the values contained in all cells that follow from the information revealed. In this note it is shown, by contrast, that the three-dimensional version of the problem is NP-complete.

It is also shown that if the suggested row, column, and file sums for an unknown three-dimensional table are given, with or without the values in some of the cells, the problem of determining whether there exists any table with the given sums is NP-complete. In the course of proving these results, the NP-completeness of some constrained Latin square construction problems, which are of some interest in their own right, is established.

Key words. data security, NP-complete problems, Latin squares

AMS subject classifications. 68R05, 68Q15, 05B15

1. Introduction.

Problems of statistical data security in two dimensions have been studied extensively—see [1] for some indications of early work and [3], [4], and [6] for some recent developments. In this note, we study the obvious extension of the problem to three dimensions, raised as an open problem by Gusfield [3], and show that, as is the case in a variety of other contexts, problems that are solvable in polynomial time in the two-dimensional case become NP-complete when extended to three dimensions.

Consider a three-dimensional table D, of size $n \times n \times n$, of nonnegative integer values $D(i, j, k)$, $(1 \leq i, j, k \leq n)$. The table entries $D(i, j, k)$ for fixed i, k and $1 \leq j \leq n$ constitute a *row* of the table, for fixed j, k and $1 \leq i \leq n$ a *column* of the table, and for fixed i, j and $1 \leq k \leq n$ a *file* of the table.

Envisage that a particular table D represents a collection of cross-tabulated statistical data, and that the row, column, and file sums of D are to be disclosed together with the values contained in some of the cells. However, other cells may contain sensitive values that are therefore to be suppressed; the question arises as to whether knowledge of the row, column, and file sums together with the disclosed cells will (a) essentially fix the values of one or more of the suppressed cells and (b) allow such values to be deduced in reasonable (say, polynomial) time (say, by an adversary). A suppressed cell that has the same value in all legal tables, i.e., all tables satisfying the row, column, and file sums and containing the disclosed values, is said to be *compromised.* A suppressed cell that is not compromised is said to be *protected.*

In the case of the corresponding two-dimensional problem, Gusfield [3] gives a $O(n^3)$ algorithm to identify all the fixed cells and to calculate their values. In [3] and [4], Gusfield also describes polynomial-time algorithms to calculate the tightest bounds on the protected cells. As part of this work, a $O(n^3)$ algorithm is presented for the identification of a legal solution in the two-dimensional case. In this context, it turns out that the obvious necessary conditions for a legal solution—namely, that the sums of the row and column sums should be equal and that each row and column sum should be at least equal to the sum of the disclosed entries in that row or column—are also sufficient for the existence of a legal solution.

[*] Received by the editors November 19, 1990; accepted for publication (in revised form) October 7, 1992.

[†] Computing Science Department, University of Glasgow, Glasgow, G12 8QQ United Kingdom.

[‡] Computer Science Department, University of Edinburgh, Edinburgh, EH9 3JZ United Kingdom.

By contrast, in the three-dimensional case, we shall show that the obvious necessary two-dimensional conditions on the row, column, and file sums are not sufficient to guarantee the existence of a legal solution, and indeed that the problem of determining whether a legal solution exists is NP-complete. This result holds even in the interesting special case in which all the cells are suppressed; in other words, if we are given the row, column, and file sums, and are asked whether a legal solution exists. We shall then proceed to show that the NP-completeness of the existence problem also implies the NP-completeness of the problem of identifying compromised cells, at least in the general case where some cells may be revealed.

The NP-completeness proofs involve consideration of special cases that are equivalent to two-dimensional problems of Latin square construction, which are of some interest in their own right. For instance, we show that a special case of the problem of the existence of a legal solution is equivalent to the problem of constructing an $n \times n$ Latin square given independent restricted choices for the various entries, and that this Latin square construction problem is itself NP-complete.

2. Formal statement of the problem. Throughout, we assume that, unless otherwise stated, the row, column, and file indices i, j, and k range over the values $1, \ldots, n$ wherever appropriate. Suppose that for a given $n \times n \times n$ table D of nonnegative integers, and for each i, j, k, the row, column, and file sums are denoted by $R(i, k)$, $C(j, k)$, and $F(i, j)$, respectively. In other words,

$$R(i,k) = \sum_{j=1}^{n} D(i,j,k), \tag{1}$$

$$C(j,k) = \sum_{i=1}^{n} D(i,j,k), \tag{2}$$

$$F(i,j) = \sum_{k=1}^{n} D(i,j,k). \tag{3}$$

If we represent the set of suppressed cells by S, i.e., $S = \{(i, j, k) : D(i, j, k)$ not disclosed$\}$, then we can calculate the reduced row, column, and file sums, namely,

$$R^*(i,k) = R(i,k) - \sum_{j:(i,j,k)\notin S} D(i,j,k), \tag{4}$$

$$C^*(j,k) = C(j,k) - \sum_{i:(i,j,k)\notin S} D(i,j,k), \tag{5}$$

$$F^*(i,j) = F(i,j) - \sum_{k:(i,j,k)\notin S} D(i,j,k). \tag{6}$$

We are now in a position to state formally the problem that we wish to pursue.

2.1. Three-dimensional statistical data—legal solutions (3DSDLS).

Instance: A positive integer n, nonnegative integer values $R^*(i, k)$, $C^*(j, k)$, and $F^*(i, j)$, and a subset S of N^3, where $N = \{1, 2, \ldots, n\}$.

Question: Does there exist an assignment of nonnegative values to $D(i, j, k)$ for $(i, j, k) \in S$ such that

$$\sum_{j=1}^{n} D(i,j,k) = R^*(i,k), \tag{7}$$

$$\sum_{i=1}^{n} D(i, j, k) = C^*(j, k), \tag{8}$$

$$\sum_{k=1}^{n} D(i, j, k) = F^*(i, j), \tag{9}$$

where each sum is taken over values $(i, j, k) \in S$?

It is immediate that, in order for a solution to exist, the sums of the R^*, C^*, and F^* values must all be identical and satisfy constraints imposed by consideration of the two-dimensional "slices" of the table, namely,

$$\sum_{k=1}^{n} R^*(i, k) = \sum_{j=1}^{n} F^*(i, j) \quad (1 \leq i \leq n), \tag{10}$$

$$\sum_{i=1}^{n} F^*(i, j) = \sum_{k=1}^{n} C^*(j, k) \quad (1 \leq j \leq n), \tag{11}$$

$$\sum_{j=1}^{n} C^*(j, k) = \sum_{i=1}^{n} R^*(i, k) \quad (1 \leq k \leq n). \tag{12}$$

To see that these necessary conditions are by no means sufficient for the existence of a legal solution, we consider an example in which $n = 2$ and all cells are suppressed.

Example. It may be checked easily by exhaustive search that the 3DSDLS instance shown in Fig. 1, in which the row, column, and file sums appear as labels on the appropriate arrows, admits no legal solution, although the necessary two-dimensional conditions are satisfied.

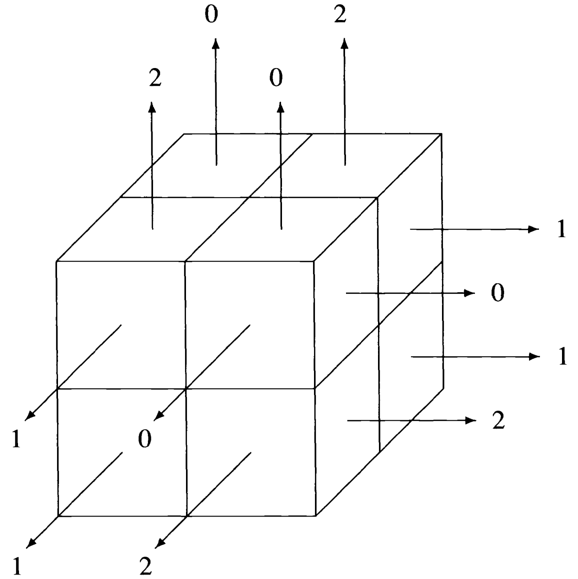


FIG. 1. *An instance of* 3DSDLS *of size* 4 *with no legal solution.*

In order to show that the 3DSDLS problem is NP-complete, we shall restrict our attention to a special case, which can be interpreted as the problem of constructing a Latin square given restricted choices for the various entries. This special case involves setting $R^*(i, k) = C^*(j, k) = F^*(i, j) = 1$ for all values of i, j, and k. If we define

$$S(i, j) = \{k : (i, j, k) \in S\},$$

then our task is to find a suppressed cell in each file, i.e., an element in $S(i, j)$, which can be given the value 1, subject to the constraint that no two cells can be chosen from the same row and no two can be chosen from the same column. It should be clear that this can be interpreted as the problem of constructing a Latin square of size $n \times n$ where the choice of element in position (i, j) is restricted to the set $S(i, j)$.

2.2. Latin square construction (LSC).

Instance: A positive integer n, and for each i, j, a subset $S(i, j)$ of $\{1, \dots, n\}$.

Question: Does there exist a Latin square X of size $n \times n$ such that, for all i, j, $X(i, j) \in S(i, j)$?

Example. The first array below constitutes a "yes"-instance of LSC of size 4, with underlined entries indicating one solution. On the other hand, exhaustive search will reveal that the second array constitutes a "no"-instance.

$$\begin{array}{llll} \{3, \underline{4}\} & \{\underline{1}, 2\} & \{1, \underline{3}, 4\} & \{1, \underline{2}\} \\ \{\underline{2}, 4\} & \{1, \underline{3}\} & \{\underline{1}, 2\} & \{3, \underline{4}\} \\ \{\underline{1}, 3\} & \{2, \underline{4}\} & \{1, \underline{2}\} & \{1, \underline{3}, 4\} \\ \{1, 2, \underline{3}\} & \{\underline{2}, 3\} & \{3, \underline{4}\} & \{\underline{1}, 2, 3\} \end{array} \qquad \begin{array}{llll} \{1, 2\} & \{1, 3\} & \{2, 4\} & \{3, 4\} \\ \{3, 4\} & \{2, 4\} & \{1, 2\} & \{1, 3\} \\ \{1, 2\} & \{3, 4\} & \{1, 3\} & \{2, 4\} \\ \{3, 4\} & \{1, 2\} & \{3, 4\} & \{1, 2\} \end{array}$$

Clearly a proof of NP-completeness for LSC implies that the more general 3DSDLS problem is NP-complete also.

3. NP-completeness of LSC. In preparation for the proof of NP-completeness of LSC, we need to investigate the conditions under which a Latin rectangle may be extended to a larger Latin rectangle or to a full Latin square. The following result is well known—see [5] or [7].

LEMMA 3.1. *An arbitrary Latin rectangle of size $m \times n$ $(m < n)$ over a ground-set of size n may be extended by the addition of $n - m$ additional rows to form a Latin square of size $n \times n$.*

In addition, we need a sufficient condition for a $p \times q$ Latin rectangle to be extendable to an $n \times q$ Latin rectangle. The following result may not be the best possible, but it will suffice for our purposes.

LEMMA 3.2. *Suppose that L is a Latin rectangle of size $p \times q$ with elements chosen from a ground-set of size n, and suppose that $n \geq p + 2q - 2$. Then L can be extended to a Latin rectangle of size $n \times q$.*

Proof. Suppose that the first j columns of L have already been extended to length n, where $0 \leq j \leq q - 1$; we show that the $(j + 1)$th column can also be extended to length n.

Consider a bipartite graph G with vertex set $V = U \cup W$. In U there are $s = n - p$ vertices, one for each of rows $p + 1, \dots, n$ in the rectangle, and in W there are n vertices, one for each element of the ground-set, which we may take to be $\{1, \dots, n\}$. Vertices $u \in U$ and $w \in W$ are joined if and only if element w already appears neither in column $j + 1$ nor in row u (and therefore w is a candidate for position (u, j) in the rectangle). It is clear that the $(j + 1)$th column can be extended to length n if and only if the graph G has a matching of size s.

By Hall's theorem, G will have a matching of size s provided that, for each k $(1 \leq k \leq s)$, the vertices in every k-subset of U are collectively adjacent to at least k vertices in W. We consider two cases:

Case (a) $k > j$. In this case, no element can appear in all the k rows corresponding to the k vertices of U. Hence, these k vertices are collectively adjacent to all n vertices in W, except for the p vertices corresponding to the elements in column $j + 1$, therefore, to $n - p$ vertices. Since $n - p \geq k$, the required condition for a matching of size s is met.

Case (b) $k \leq j$. In this case, it is possible that the k rows have up to k elements in common, so that the best we can claim is that the k vertices in U are collectively adjacent to at least $n - p - k$ vertices in W. Nonetheless, since $n - p \geq 2(q - 1)$, $q - 1 \geq j$, and $j \geq k$, it follows that $n - p - k \geq k$, and the required condition is again satisfied.

Hence, by Hall's theorem, G has a matching of size s, and therefore the $(j + 1)$th column can be extended to length n as claimed. □

We are now in a position to prove the NP-completeness of our Latin square construction problem LSC.

THEOREM 3.3. *Latin square construction is NP-complete.*

Proof. Membership in NP is immediate, for we need simply guess an element $X(i, j) \in S(i, j)$ for each i, j, and it is straightforward to verify, in polynomial time, whether the resulting square X is a Latin square.

To show that LSC is NP-complete, we describe a polynomial-time transformation from the known NP-complete problem 3-Satisfiability (3-SAT)—see [2].

Given an instance of 3-SAT involving m variables $v_1, v_2, \ldots, v_m$ and n clauses $C_1, C_2, \ldots, C_n$, we construct an instance of LSC of size $2mn$, which admits a Latin square if and only if the original 3-SAT instance is satisfiable. Corresponding to each v_k there are precisely $2n$ elements in the ground-set S for the derived instance of LSC, denoted by $u_{kl}, \overline{u}_{kl}$ $(1 \leq l \leq n)$. It remains to describe the sets $S(i, j)$ $(1 \leq i, j \leq 2mn)$.

For $1 \leq k \leq m$, $1 \leq l \leq n$, we define

$$\begin{aligned} S((k-1)n + l, 1) &= \{u_{kl}, \overline{u}_{kl}\}, \\ S((k-1)n + l, 2) &= \{u_{kl}, u_{k,l+1}\}, \\ S((k-1)n + l, 3) &= \{u_{k,l+1}, \overline{u}_{k,l}\}, \end{aligned}$$

where here, and subsequently, $l + 1$ is taken modulo n in the range $[1, n]$.

For $1 \leq l \leq n$ we define

$$S(mn + l, 1) = \{w_{1l}, w_{2l}, w_{3l}\},$$

where

$$w_{hl} = \begin{cases} u_{kl} & \text{if the } h\text{th literal in } C_l \text{ is } v_k, \\ \overline{u}_{kl} & \text{if the } h\text{th literal in } C_l \text{ is } \overline{v}_k. \end{cases}$$

Finally, for all subscript pairs i, j not covered by the above, we set $S(i, j) = S$. It is clear that the entire construction can be carried out in time bounded by a polynomial in the length of the original 3-SAT expression.

We now have to establish that the derived instance of LSC admits the construction of a $2mn \times 2mn$ Latin square if and only if the original instance of 3-SAT has a satisfying assignment.

Suppose first that the LSC instance admits a Latin square X. For a given value of k, consider positions $(k - 1)n + l, t$ $(1 \leq l \leq n, 1 \leq t \leq 3)$ in the square. It is straightforward to verify that there are just two possibilities, either

$$\text{(a)} \qquad \begin{aligned} X((k-1)n+l,1) &= \overline{u}_{kl},\\ X((k-1)n+l,2) &= u_{kl},\\ X((k-1)n+l,3) &= u_{k,l+1} \end{aligned}$$

for all l $(1 \leq l \leq n)$, or

$$\text{(b)} \qquad \begin{aligned} X((k-1)n+l,1) &= u_{kl},\\ X((k-1)n+l,2) &= u_{k,l+1},\\ X((k-1)n+l,3) &= \overline{u}_{kl} \end{aligned}$$

for all l $(1 \leq l \leq n)$.

We assign variable v_k to be true or false accordingly as Case (a) or Case (b) applies.

Now, for each l, $1 \leq l \leq n$, we consider $X(mn+l,\, 1)$. Because X is a Latin square, we must have $X(mn+l,\, 1)$ equal to w_{hl}, where w_{hl} represents a true literal; otherwise, by (a) and (b) above, we would have $X(mn+l,\, 1) = X((k-1)n+l,\, 1)$ for some k. So, in the derived assignment, every clause contains at least one true literal, showing that the original instance of 3-SAT is satisfiable.

On the other hand, suppose that the instance of 3-SAT is satisfiable, and consider a particular satisfying assignment. If v_k is true in this assignment, we choose

$$\begin{aligned} X((k-1)n+l,1) &= \overline{u}_{kl},\\ X((k-1)n+l,2) &= u_{k,l},\\ X((k-1)n+l,3) &= u_{k,l+1} \end{aligned}$$

for all l $(1 \leq l \leq n)$, while if v_k is false, we choose

$$\begin{aligned} X((k-1)n+l,1) &= u_{kl},\\ X((k-1)n+l,2) &= u_{k,l+1},\\ X((k-1)n+l,3) &= \overline{u}_{kl} \end{aligned}$$

for all l $(1 \leq l \leq n)$.

Further, for $1 \leq l \leq n$, we choose

$$X(mn+l,1) = w_{hl},$$

where w_{hl} is a true literal in clause C_l, and

$$\begin{aligned} X(mn+l,2) &= \overline{u}_{1,l+1},\\ X(mn+l,3) &= \begin{cases} u_{1,l+1} & \text{if } v_1 \text{ is false},\\ \overline{u}_{1,l+2} & \text{if } v_1 \text{ is true}. \end{cases} \end{aligned}$$

It is straightforward to verify that this gives a Latin rectangle of size $(m+1)n \times 3$. Provided $(m-1)n \geq 4$, which can be assumed without loss of generality, the condition of Lemma 3.2

is satisfied; this Latin rectangle can then be extended, first, to form a Latin rectangle of size $2mn \times 3$, and then, by Lemma 3.1, to form a Latin square of size $2mn \times 2mn$, bearing in mind that $S(i, j) = S$ for all outstanding positions (i, j). □

In view of the earlier observation that LSC is a special case of 3DSDLS, we have the following corollary.

COROLLARY 3.4. 3DSDLS *is NP-complete, even in the special case where all the row, column and file sums are* 1.

4. The special case of all cells suppressed. We now consider the interesting special case of 3DSDLS in which all cells are suppressed. This special case is a natural problem in its own right, which we refer to as the 3D contingency table problem (3DCT).

4.1 Three-dimensional contingency tables (3DCT).

Instance: A positive integer n, and for each i, j, k, nonnegative integer values $R(i, k)$, $C(j, k)$, $F(i, j)$.

Question: Does there exist an $n \times n \times n$ contingency table X of nonnegative integers such that

$$\textstyle\sum_{j=1}^{n} X(i, j, k) = R(i, k),$$

$$\textstyle\sum_{i=1}^{n} X(i, j, k) = C(j, k),$$

$$\textstyle\sum_{k=1}^{n} X(i, j, k) = F(i, j)$$

for all i, j, k?

We now establish the NP-completeness of 3DCT even in the special case where all the R, C, and F values are 0 or 1. This we achieve by viewing this special case as a two-dimensional partial Latin square construction problem, and by showing that there is a polynomial-time transformation from the basic Latin square construction problem, already known to be NP-complete, to this new problem.

The partial Latin square construction problem is specified in the following section.

4.2 Partial Latin square construction (PLSC).

Instance: A positive integer n, subsets $R(i)$ and $C(j)$ of $N = \{1, \ldots, n\}$ for each i, j, and a subset $\mathcal{N}$ of N^2 such that (a) $|R(i)| = |k : (i, k) \in \mathcal{N}|$, and (b) $|C(j)| = |k : (k, j) \in \mathcal{N}|$.

Question: Does there exist a partial Latin square X with

(i) $X(i, j)$ defined for all $(i, j) \in \mathcal{N}$;

(ii) $X(i, j) \in R(i) \cap C(j)$ for all such (i, j)?

Note that it follows at once that, in a "yes" instance of PLSC, (a) $k \in R(i) \Rightarrow X(i, j) = k$ for some j, and (b) $k \in C(j) \Rightarrow X(i, j) = k$ for some i.

Example. In Fig. 2, the sets $R(i)$ and $C(j)$ appear to the left of the rows and above the columns, respectively; the set $R(i) \cap C(j)$ appears in position (i, j); cells corresponding to pairs $(i, j) \notin \mathcal{N}$ contain the symbol —; and the underlined elements indicate that this is a "yes"-instance of the problem.

Our first objective is to show that the special case of 3DCT in which the row, column, and file sums are all 0 or 1 can be interpreted as an instance of PLSC. Given such an instance of 3DCT of size n with row, column, and file sums $R(i, k)$, $C(j, k)$, and $F(i, j)$, define

$$\begin{aligned} \mathcal{N} &= \{(i, j) : F(i, j) = 1\}, \\ R(i) &= \{k : R(i, k) = 1\}, \\ C(j) &= \{k : C(j, k) = 1\}. \end{aligned}$$

	$\{1,3\}$	$\{1,2,4\}$	$\{3,4\}$	$\{1,2,3,4\}$
$\{1,2,3\}$	—	$\underline{1}, 2$	$\underline{3}$	$1, \underline{2}, 3$
$\{1,4\}$	$\underline{1}$	—	—	$1, \underline{4}$
$\{2,3,4\}$	—	$\underline{2}, 4$	$3, \underline{4}$	$2, \underline{3}, 4$
$\{1,3,4\}$	$1, \underline{3}$	$1, \underline{4}$	—	$\underline{1}, 3, 4$

FIG. 2. *An instance of* PLSC *of size* 4.

The set $\mathcal{N}$ represents the set of files with sum equal to 1. The sole position in such a file, say, file (i, j), occupied by a 1 must correspond to both a row with sum equal to 1—i.e., an element of $R(i)$—and a column with sum equal to 1—i.e., an element of $C(j)$. It follows that a solution to the given instance of the special case of 3DCT corresponds exactly to a solution of the derived instance of PLSC.

We are now in a position to establish the NP-completeness of PLSC.

THEOREM 4.1. *The partial Latin square construction problem* (PLSC) *is* NP-*complete.*

Proof. Membership in NP is immediate, since we can guess an element in the ground-set for each cell in the set $\mathcal{N}$ and verify in polynomial time that the various row and column constraints are satisfied. To prove NP-completeness, we describe a polynomial-time transformation from the basic Latin square construction problem (LSC), already known to be NP-complete by Theorem 3.3.

Given a positive integer n and sets $S(i, j) \subseteq N = \{1, \ldots, n\}$ forming an instance of LSC, we construct an instance of PLSC over the same ground-set N but of increased size $n' = n + \sum_{ij} |S(i, j)|$. The set of free cells $\mathcal{N}$ is defined as a disjoint union of n^2 component sets $\mathcal{N}_{ij}$, which are constructed as follows. Let λ, μ be mappings from N^2 to $\{n + 1, n + 2, \ldots, n'\}$ satisfying

(i) $\mu(i, j) - \lambda(i, j) + 1 = |S(i, j)|$, for all $i, j \in N$;

(ii) the intervals $\{[\lambda(i, j), \mu(i, j)] \; : \; i, j \in N\}$ form a partition of the set $\{n + 1, n + 2, \ldots, n'\}$. (The notation $[a, b]$ denotes the set of all integers not less than a and not greater than b.)

For each i, j in the range $1 \leq i, j \leq n$, define

$$\mathcal{N}_{ij} = [l, m] \times [l, m] \cup \{(i, l), (l, j)\} - \{(l, l)\},$$

where $l = \lambda(i, j)$ and $m = \mu(i, j)$; the set of free cells is then $\mathcal{N} = \bigcup_{ij} \mathcal{N}_{ij}$. Figure 3 is a pictorial representation of a typical component set $\mathcal{N}_{ij}$.

We now specify the row and column sets. For k in the range $1 \leq k \leq n$, set $R(k) = C(k) = N$. For k outside this range, i.e., for $n+1 \leq k \leq n'$, let $i, j \in N$ be the unique integers satisfying $\lambda(i, j) \leq k \leq \mu(i, j)$, and set $R(k) = C(k) = S(i, j)$. It can be checked that R, C, and $\mathcal{N}$ together form a consistent instance of PLSC. This completes the construction.

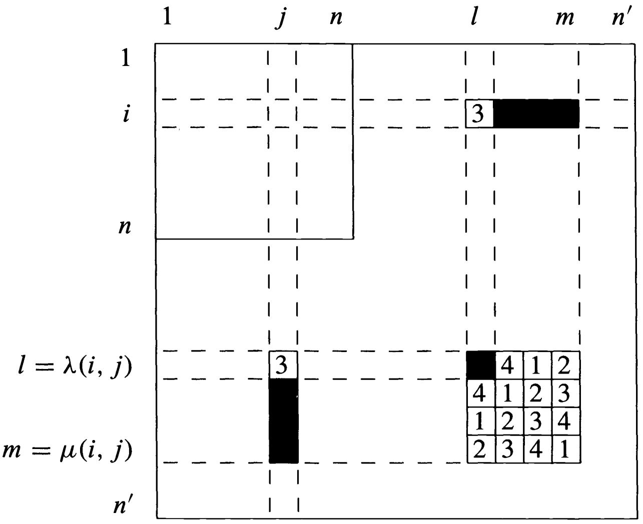


FIG. 3. *The set $\mathcal{N}_{ij}$, and a possible numbering when $S(i, j) = \{1, 2, 3, 4\}$.*

To verify the construction, we must show that the derived instance of PLSC admits a solution if and only if the original instance of LSC does. Suppose first that the LSC instance admits a Latin square X, so that $X(i, j) \in S(i, j)$ for all i, j. We shall construct a partial Latin square Y consistent with the derived instance of PLSC, i.e., satisfying $Y(i, j) \in R(i) \cap C(j)$ for all $(i, j) \in \mathcal{N}$.

It is enough to describe the restriction of Y to a typical component set $\mathcal{N}_{ij}$, since $\mathcal{N}$ is a disjoint union of such sets. Let $l = \lambda(i, j)$, $m = \mu(i, j)$, and $s = |S(i, j)|$. We consider the domain $\mathcal{N}_{ij}$ in two parts: the square with missing corner cell $[l, m] \times [l, m] - \{(l, l)\}$, and the two isolated cells (i, l) and (l, j). On the isolated cells, we simply take $Y(i, l) = Y(l, j) = X(i, j)$. In order to deal with the square, imagine that the missing corner cell is temporarily reinstated. Define Y on this completed square so that the resulting $s \times s$ table of values forms a Latin square over the ground-set $S(i, j)$, with $Y(l, l) = X(i, j)$. Then simply remove the cell (l, l) from the domain of definition of Y. Figure 3 is intended to illustrate how the restriction of Y to the set $\mathcal{N}_{ij}$ might appear in the case $S(i, j) = \{1, 2, 3, 4\}$.

It may readily be checked that $Y(i, j) \in R(i) \cap C(j)$ for all cells $(i, j) \in \mathcal{N}$. The only other condition we must verify is that Y is indeed a partial Latin square, i.e., that no row or column of Y contains a duplicate value. By symmetry, we need only check this condition for the rows. For i in the range $n + 1 \leq i \leq n'$, the fact that row i of Y cannot contain duplicate values is clear by construction. So suppose $1 \leq i \leq n$, which is the only other possibility. Row i of the derived instance of PLSC contains precisely n cells that are elements of $\mathcal{N}$, and these correspond to the n cells forming row i in the original instance of LSC. Y assigns to each of the n cells in the PLSC instance the same value that X assigns to the corresponding cell in the LSC instance, and these n values must all be distinct. Thus we have shown that the PLSC instance admits a solution if the LSC instance does.

For the converse, suppose that the derived PLSC instance admits a partial Latin square Y that is consistent with the various row and column constraints. We shall construct a Latin square X consistent with the original instance of LSC. Consider the restriction of Y to the set $\mathcal{N}_{ij}$, for some $1 \leq i, j \leq n$. As before, let $l = \lambda(i, j)$ and $m = \mu(i, j)$. The restriction of Y to the array of cells $[l, m] \times [l, m]$ forms a Latin square on ground-set $S(i, j)$ with the top left corner cell removed. The values $Y(i, l)$ and $Y(l, j)$ must both equal the missing element of this Latin square, and hence $Y(i, l) = Y(l, j) \in S(i, j)$. Now set $X(i, j)$ equal to $Y(i, l)$.

Since the individual cell constraints are clearly satisfied, it only remains to show that X is indeed a Latin square, i.e., that no row or column of X contains duplicate values. By symmetry, we need only check this condition for rows. But the n values occurring in the ith row of X are equal, by construction, to the n corresponding values occurring in the ith row of Y. Hence the ith row of X cannot contain duplicate values. We have thus demonstrated that the LSC instance admits a solution if the PLSC instance does. This completes the validation of the reduction. □

COROLLARY 4.2. 3DCT *is* NP-*complete, even in the special case where all the row, column and file sums are* 0 *or* 1.

5. NP-completeness of identifying compromised cells. We now consider the question of identifying the compromised cells in the general three-dimensional problem. In other words, if we are given a particular legal solution to an instance of 3DSDLS, and we focus attention on a particular (suppressed) cell, we wish to establish whether there exists a second solution in which that cell holds a different value. We shall show that determining whether a particular cell is compromised is also an NP-complete problem.

First, we give a formal description of the problem.

5.1. Three-dimensional statistical data—compromised cells (3DSDCC).

Instance: A positive integer n, a subset S of N^3, where $N = \{1, \dots, n\}$, nonnegative integer values $D(i, j, k)$ for each $(i, j, k) \in S$, and a particular triple $(i_0, j_0, k_0) \in S$.

Question: Does there exist a set of nonnegative integer values $D'(i, j, k)$ $((i, j, k) \in S)$ such that

(i) $\sum_{i=1}^n D'(i, j, k) = \sum_{i=1}^n D(i, j, k)$,

(ii) $\sum_{j=1}^n D'(i, j, k) = \sum_{j=1}^n D(i, j, k)$,

(iii) $\sum_{k=1}^n D'(i, j, k) = \sum_{k=1}^n D(i, j, k)$,

and

(iv) $D'(i_0, j_0, k_0) \neq D(i_0, j_0, k_0)$,

where all sums are taken over $(i, j, k) \in S$?

As was the case with our earlier problem, we prove 3DSDCC NP-complete by considering a special case that we can interpret in terms of Latin squares. Again, this is the special case in which all the row, column, and file sums are equal to 1. As earlier, we can interpret any solution to such an instance as a Latin square X of size $n \times n$ in which $X(i, j) \in S(i, j)$, where $S(i, j) = \{k : (i, j, k) \in S\}$.

In fact, the NP-completeness of the Latin square nonuniqueness problem that we are about to describe has a slightly stronger consequence than we need, namely, that the problem of determining whether there is any legal solution, other than the one given, is NP-complete. The NP-completeness of 3DSDCC follows at once from this, for if we had a polynomial-time algorithm for the latter problem, we could apply it at most a polynomial number of times to determine whether there is any other legal solution.

5.2. Latin square nonuniqueness (LSNU).

Instance: A positive integer n, subsets $S(i, j)$ of $N = \{1, \dots, n\}$ for each i,j, and a Latin square X of size $n \times n$ with $X(i, j) \in S(i, j)$ for all i, j.

Question: Does there exist a Latin square X' of size $n \times n$ such that $X'(i, j) \in S(i, j)$ for all i, j, and $X' \neq X$?

As observed above, the NP-completeness of 3DSDCC will follow from the NP-completeness of LSNU. Before proving this result, we need some further notation and a lemma.

For arbitrary values of i, j $(1 \leq i, j \leq 2n)$, we write $p(i, j)$ for the value of $i + j - 1$ taken mod$n$ in $[1, n]$, so that, clearly, $p(i, j + n) = p(i, j)$.

LEMMA 5.1. *For fixed i, let*

$$S(j) = \begin{cases} \{p(i,j), p(i,j)+n\} & (1 \le j \le n), \\ \{p(i,j), p(i,j+1)+n\} & (n+1 \le j \le 2n). \end{cases}$$

Then the sets $S(j)$ $(1 \le j \le 2n)$ have exactly two sets of distinct representatives, namely, $s(j)$ and $t(j)$ defined by

$$s(j) = \begin{cases} p(i,j) & (1 \le j \le n), \\ p(i,j+1)+n & (n+1 \le j \le 2n) \end{cases}$$

and

$$t(j) = \begin{cases} p(i,j)+n & (1 \le j \le n), \\ p(i,j) & (n+1 \le j \le 2n). \end{cases}$$

Proof. First of all we observe the following intersections:

$$\begin{aligned} S(j) \cap S(j+n) &= \{p(i,j)\} && (1 \le j \le n), \\ S(j+1) \cap S(j+n) &= \{p(i,j+1)+n\} && (1 \le j \le n), \end{aligned}$$

where $j+1$ is taken mod$n$ in $[1,n]$. So, if we choose $p(i,1)$ as the representative for $S(1)$, we are forced to choose $p(i,2)+n$ as the representative for $S(n+1)$, which in turn forces us to choose $p(i,2)$ as the representative for $S(2)$, and so on, leading to the system s defined above. On the other hand, if we choose $p(i,1)+n$ as the representative for $S(1)$, we are forced to choose $p(i,2n)$ as the representative for $S(2n)$, which in turn forces us to choose $p(i,n)+n$ as the representative for $S(n)$, and so on, leading to the system t defined above. These are the only two possibilities. □

The following further lemma is analogous to the previous one.

LEMMA 5.2. *For fixed j, let*

$$S(i) = \begin{cases} \{p(i,j), p(i,j)+n\} & (1 \le i \le n), \\ \{p(i,j), p(i+1,j)+n\} & (n+1 \le i \le 2n). \end{cases}$$

Then the sets $S(i)$ $(1 \le i \le 2n)$ have exactly two sets of distinct representatives, namely, $s(i)$ and $t(i)$ defined by

$$s(i) = \begin{cases} p(i,j) & (1 \le i \le n), \\ p(i+1,j)+n & (n+1 \le i \le 2n) \end{cases}$$

and

$$t(i) = \begin{cases} p(i,j)+n & (1 \le i \le n), \\ p(i,j) & (n+1 \le i \le 2n). \end{cases}$$

THEOREM 5.3. *Latin square nonuniqueness is NP-complete.*

Proof. Membership in NP is immediate. We need merely guess values $X'(i, j) \in S(i, j)$ for each (i, j), and easily verify in polynomial time that X' is a Latin square, and that $X' \neq X$.

To prove NP-completeness, we describe a polynomial-time transformation from LSC, known to be NP-complete by Theorem 3.3, to LSNU.

Given an instance of LSC of size n with ground-set $\{1, \ldots, n\}$ and subsets $S(i, j)$ $(1 \leq i, j \leq n)$, we construct from it an instance of LSNU of size $2n$, with ground-set $\{1, \ldots, 2n\}$ and subsets $S'(i, j)$ $(1 \leq i, j \leq 2n)$ as follows.

We set

$$S'(i, j) = \begin{cases} S(i, j) \cup \{p(i, j) + n\} & \text{for } 1 \leq i, j \leq n\,, \\ \{p(i, j), p(i, j) + n\} & \text{for } n + 1 \leq i \leq 2n, 1 \leq j \leq n, \\ & \text{and } 1 \leq i \leq n, n + 1 \leq j \leq 2n, \\ \{p(i, j), p(i - 1, j) + n\} & \text{for } n + 1 \leq i, j \leq 2n. \end{cases}$$

The Latin square for this instance of LSNU is defined by

$$X(i, j) = \begin{cases} p(i, j) + n & \text{for } 1 \leq i, j \leq n\,, \\ p(i, j) & \text{for } n + 1 \leq i \leq 2n, 1 \leq j \leq n, \\ & \text{and } 1 \leq i \leq n, n + 1 \leq j \leq 2n, \\ p(i - 1, j) + n & \text{for } n + 1 \leq i, j \leq 2n. \end{cases}$$

Verification that X is a Latin square and that $X(i, j) \in S(i, j)$ for all i, j is straightforward.

We now show that there is a Latin square X' with $X'(i, j) \in S(i, j)$ for all i, j, and X' nonidentical to X if and only if the original instance of LSC is solvable, and indeed if this is the case then $X'(i, j) \neq X(i, j)$ for all i, j.

First of all, if the LSC instance is solvable, with $n \times n$ Latin square Y, $Y(i, j) \in S(i, j)$ $(1 \leq i, j \leq n)$, we may choose

$$X'(i, j) = \begin{cases} Y(i, j) & \text{for } 1 \leq i, j \leq n, \\ p(i, j) + n & \text{for } n + 1 \leq i \leq 2n, 1 \leq j \leq n, \\ & \text{and } 1 \leq i \leq n, n + 1 \leq j \leq 2n, \\ p(i, j) & \text{for } n + 1 \leq i, j \leq 2n. \end{cases}$$

to give a $2n \times 2n$ Latin square X', with $X'(i, j) \neq X(i, j)$ for all i, j.

On the other hand, let us consider the circumstances under which a different $2n \times 2n$ Latin square may exist. If for some (i_0, j_0), $X'(i_0, j_0) \neq X(i_0, j_0)$, then it is immediate that $X'(i', j') \neq X(i', j')$ for some (i', j') with $1 \leq i' \leq n, n + 1 \leq j' \leq 2n$. By Lemma 5.1, it follows that $X'(i, j) \neq X(i, j)$ for all i, j $(n + 1 \leq i \leq 2n, 1 \leq j \leq 2n)$. A further application of Lemma 5.2 reveals that this is true also for $1 \leq i \leq n, n + 1 \leq j \leq 2n$. Hence, in order that X' may be a Latin square, every value $X'(i, j)$ $(1 \leq i, j \leq n)$ must belong to $S(i, j)$, and therefore these values constitute a Latin square for the original instance of LSC. □

Finally, we consider the special case in which all cells are suppressed. This version of the problem can be expressed as described in the following section.

5.3. Contingency table nonuniqueness (CTNU).

Instance: A positive integer n and an $n \times n \times n$ table X of nonnegative integers.

Question: Does there exist an $n \times n \times n$ table $X'(\neq X)$ such that X' and X have identical row, column, and file sums?

Superficially, this version of the problem might appear easier than the general 3DS-DCC problem. For example, the existence of nonzero entries in positions (i, j, k'), (i, j', k), (i', j, k), and (i', j', k') for some i, i', j, j', k, k' would constitute an obvious sufficient condition for a "yes" answer—these entries could be reduced by 1 and the entries in positions (i, j, k), (i, j', k'), (i', j, k'), and (i', j', k) increased by 1 to give a second legal solution.

However, contingency table nonuniqueness is still NP-complete, as we now show. We first of all observe, in the same spirit as previously, that the special case of CTNU in which all row, column and file sums are 0 or 1 can be phrased as a partial Latin square problem, as follows.

5.4. Partial Latin square nonuniqueness (PLSNU).

Instance: A positive integer n, subsets $R(i)$ and $C(j)$ of $N = \{1, \ldots, n\}$ for each i, j, a subset $\mathcal{N}$ of N^2 such that (a) $|R(i)| = |k : (i, k) \in \mathcal{N}|$, and (b) $|C(j)| = |k : (k, j) \in \mathcal{N}|$, and values $X(i, j)$ for all $(i, j) \in \mathcal{N}$ satisfying

(i) $X(i, j) \in R(i) \cap C(j)$;

(ii) $i \neq i' \Rightarrow X(i, j) \neq X(i', j)$;

(iii) $j \neq j' \Rightarrow X(i, j) \neq X(i, j')$.

Question: Do there exist values $X'(i, j)$ for all $(i, j) \in \mathcal{N}$ satisfying (i), (ii), and (iii) above, such that $X'(i_0, j_0) \neq X(i_0, j_0)$ for some i_0, j_0?

THEOREM 5.4. CTNU *is* NP-*complete.*

Proof. Membership in NP is obvious, since we can guess X' and easily verify the required conditions in polynomial time.

The key to the proof of NP-completeness is the observation that the polynomial-time transformation from LSC to PLSC given in the proof of Theorem 4.1 can, with relatively minor adjustments, be made parsimonious, i.e., so that there is a one-to-one correspondence between solutions to the original LSC instance and solutions to the derived PLSC instance. Hence, this parsimonious version gives us a polynomial-time transformation from LSNU to PLSNU, and in view of Theorem 5.3, establishes the NP-completeness of the latter problem. It remains to describe the details of this parsimonious transformation.

Define n', λ, and μ as in the proof of Theorem 4.1; recall that n' denotes the size of the derived instance of PLSC. As before, the set $\mathcal{N}$ of free cells is constructed as a disjoint union of component sets $\mathcal{N}_{ij}$. For each i, j in the range $1 \leq i, j \leq n$ define

$$\mathcal{N}_{ij} = \{(k, l), (k, k), (l, k) : l + 1 \leq k \leq m\} \cup \{(i, l), (l, j)\},$$

where $l = \lambda(i, j)$ and $m = \mu(i, j)$. (Note that at this point the construction diverges somewhat from that employed in the proof of Theorem 4.1.) As before we define $\mathcal{N} = \bigcup_{ij} \mathcal{N}_{ij}$.

We now specify the row and column sets. For k in the range $1 \leq k \leq n$, set $R(k) = C(k) = N$. For k outside this range, i.e., for $n + 1 \leq k \leq n'$, let $i, j \in N$ be the unique integers satisfying $\lambda(i, j) \leq k \leq \mu(i, j)$. Let $l = \lambda(i, j)$, $m = \mu(i, j)$, $s = |S(i, j)|$, and let $\{v_1, \ldots, v_s\}$ be an enumeration of the elements of the set $S(i, j)$. Then define

$$R(k) = C(k) = \begin{cases} S(i, j), & \text{if } k = l; \\ \{v_{k-l}, v_{k-l+1}\}, & \text{otherwise.} \end{cases}$$

It can be checked that R, C, and $\mathcal{N}$ together form a consistent instance of PLSC. This completes the construction. Figure 4 is intended to illustrate a fragment of the derived instance of

PLSC, and features the rows and columns that lie within the range $[l, m]$. (The conventions employed here are the same as those of Fig. 2.) For this example, we have again taken $S(i, j) = \{1, 2, 3, 4\}$. The intention is that this 4×4 square should replace the 4×4 square at the bottom-right corner of Fig. 3.

	$\{1,2,3,4\}$	$\{1,2\}$	$\{2,3\}$	$\{3,4\}$
$\{1,2,3,4\}$	—	$\underline{1}, 2$	$\underline{2}, 3$	$3, \underline{4}$
$\{1,2\}$	$\underline{1}, 2$	$1, \underline{2}$	—	—
$\{2,3\}$	$\underline{2}, 3$	—	$2, \underline{3}$	—
$\{3,4\}$	$3, \underline{4}$	—	—	$\underline{3}, 4$

FIG. 4. *A fragment of the derived instance of* PLSC.

The verification of the reduction relies on the following observation. Let Y be any partial Latin square that is consistent with the derived instance of PLSC. Consider the restriction of Y to the set $\mathcal{N}_{ij}$. Recall that $\{v_1, \dots, v_s\}$ is an enumeration of the set $S(i, j)$, and suppose that $Y(i, l) = v_t$. Observe that column l of Y is completely forced: from cell $(l+1, l)$ to cell (m, l) the entries must read $v_1, v_2, \dots, v_{t-1}, v_{t+1}, v_{t+2}, \dots, v_s$; these entries constrain the diagonal, which from cell $(l + 1, l + 1)$ to cell (m, m) must read $v_2, v_3, \dots, v_t, v_t, v_{t+1}, \dots, v_{s-1}$; these in turn constrain row l, which from cell $(l, l + 1)$ to (l, m) must read $v_1, v_2, \dots, v_{t-1}$, $v_{t+1}, v_{t+2}, \dots, v_s$. Figure 4 illustrates the pattern that emerges when $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, $v_4 = 4$, and $t = 3$.

It follows from this chain of reasoning that $Y(l, j) = v_t = Y(i, l)$. Moreover, once $Y(i, j)$ has been chosen, there is precisely one way to extend Y to the remaining cells in $\mathcal{N}_{ij}$. Armed with this observation, the validation of the reduction proceeds as in the proof of Theorem 4.1; the reduction is clearly parsimonious. □

6. The case of "slice" sums. In an alternative extension to three dimensions of the original two-dimensional problem, suppose that we are given the sums of all two-dimensional "slices" of a three-dimensional table rather than the sums of the one-dimensional rows, columns, and files. In practice, for example, we might be given the row and column sums for three two-dimensional tables relating each pair of three distinct attributes of a population. Such a population could be represented in a three-dimensional table relating all three attributes, and the row and column sums for the two-dimensional tables would translate into the sums of the two-dimensional slices in the three-dimensional table.

The question arises as to the status, in this context, of problems corresponding to those investigated in earlier sections when row, column, and file sums are given. We conclude by summarizing the main results that hold for this version of the problem. They are again NP-completeness results, with the notable exception of the contingency table problem.

Consider an $n \times n \times n$ table D of nonnegative integer values $D(i, j, k)$, and define the slice sums

$$X(i) = \sum_{j,k} D(i, j, k),$$

$$Y(j) = \sum_{i,k} D(i, j, k),$$

$$Z(k) = \sum_{i,j} D(i, j, k).$$

The following theorem and corollary are analogous to the results established by Gusfield in [3, §5] and can be proved by similar methods, with a simple induction argument replacing the use of network flow.

THEOREM 6.1. *Suppose that, for each i, j, k, we are given nonnegative integer values $X(i)$, $Y(j)$, $Z(k)$ such that $\sum_i X(i) = \sum_j Y(j) = \sum_k Z(k) = T$. Then there exists a table D for which $X(i)$, $Y(j)$, and $Z(k)$ are the slice sums. Furthermore, the tightest upper and lower bounds on the value of $D(i, j, k)$ are* $\min(X(i), Y(j), Z(k))$ *and* $\max(0, X(i) + Y(j) + Z(k) - 2T)$, *respectively.*

COROLLARY 6.2. *In the context of the previous theorem, only the cells in the slice with the largest sum can have a nonzero lower bound, and this can happen only if that slice has sum $> 2T/3$.*

The results of the above theorem contrast with the NP-completeness of the corresponding problem when row, column, and file sums are given (3DCT) and perhaps give us some hope that, when the values of some cells are revealed as well as the slice sums, there may be polynomial-time algorithms to determine a legal solution and to identify compromised cells. However, this turns out not to be so.

As far as finding a legal solution is concerned, it is not hard to see that the special case in which all slice sums over suppressed cells are equal to 1 is just the three-dimensional matching problem, which is well known to be NP-complete [2].

Finally, the problem of identifying compromised cells in this context can be proved NP-complete by a reduction and argument similar to, but easier than, that used in the proof of Theorem 5.3.

REFERENCES

[1] D. DENNING, *Cryptography and Data Security*, Addison-Wesley, Reading, MA, 1982.

[2] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, Freeman, San Francisco, 1979.

[3] D. GUSFIELD, *A graph theoretic approach to statistical data security*, SIAM J. Comput., 17 (1988), pp. 552–571.

[4] ———, *Faster dection of compromised data in 2-D tables*, Tech. Report CSE-89-30, Computer Science Division, University of California, Davis, CA, 1989.

[5] M. HALL, JR., *Combinatorial Theory*, Blaisdell, Waltham, MA, 1967.

[6] M.-Y. KAO AND D. GUSFIELD, *Efficient detection and protection of information in cross tabulated tables: Linear invariant test*, SIAM J. Disc. Math., 6 (1993), pp. 460–476.

[7] C. L. LIU, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 185–211, February 1994

013

AN NC ALGORITHM FOR SCHEDULING UNIT-TIME JOBS WITH ARBITRARY RELEASE TIMES AND DEADLINES*

GREG N. FREDERICKSON† AND SUSAN H. RODGER‡

Abstract. The problem of scheduling n unit-time jobs with real-valued release times and deadlines is shown to be in NC. The solution is based on characterizations of a canonical schedule and best subset of jobs to be scheduled in a given time interval. The algorithm runs on a CREW PRAM in $O((\log n)^2)$ time and uses $O(n^4/\log n)$ processors.

Key words. parallel algorithm, scheduling, release time, deadline

AMS subject classifications. 68Q22, 68R05, 90B35

1. Introduction. A major goal in the study of parallel algorithms is the elucidation of the underlying combinatorial structure of problems. A wealth of insight has been generated by designing parallel algorithms for problems in such areas as graph theory, algebra and arithmetic, and computational geometry [KR]. To a lesser extent, parallel algorithms have been reported in scheduling theory [DUW], [HM1], [HM2]. In this paper we focus on a fundamental problem in scheduling theory and identify significant features of it that lead to an NC algorithm. The problem is to find a schedule for a set of jobs on a single processor, where the jobs each have unit-time processing requirements and real-valued release times and deadlines.

Our problem is intermediate in conceptual difficulty between the following two variations. The first variation has integral release times and deadlines in addition to unit processing times, and can be solved sequentially by the earliest-deadline-first rule [H] and in parallel by a parallel implementation of this rule [AGK], [DS], [R]. The second variation has unequal processing times, and has been shown to be NP-complete [GJ]. Our problem was posed as open in [GJ] as to whether it is polynomially solvable or NP-complete. It was shown in [C], [S1] that the problem is polynomially solvable, and an $O(n \log n)$-time algorithm was presented in [GJST]. Polynomial-time algorithms for finding a multiprocessor schedule were given in [S2] and [SW]. These approaches appear inherently sequential, and the problem is challenging in a parallel regimen for the following reason. Since the release times and deadlines are arbitrary real values, the appropriate scheduling choice at a given time might be to schedule no job and allow some fraction of idle time until another job's release time is reached. These choices can be affected by jobs whose release times and deadlines are far from such a decision point, making it difficult to resolve such choices "locally." We show how to work around this difficulty, and present an algorithm that uses $O((\log n)^2)$ time and $O(n^4/\log n)$ processors on a CREW PRAM.

We sketch our approach briefly and indicate the features that make this approach possible. The set of jobs is partitioned into subsets based on their release times, such that each subset has associated with it a time interval, which contains the release times of the jobs in the subset. For each such interval, a "best" set of jobs is tentatively chosen, from among those jobs assigned to the interval, to be scheduled within the interval. This best set is such that the jobs that are not chosen have the largest deadlines among all such sets. (These notions are defined precisely

*Received by the editors August 20, 1990; accepted for publication (in revised form) October 8, 1992.

†Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by National Science Foundation grants CCR-86202271 and CCR-9001241 and the Office of Naval Research contract N00014-86-K-0689.

‡Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180-3590. The research of this author was supported by the Office of Naval Research on contract N 00014-86-K-0689. The major portion of this research was done while this author was at Purdue University.

later.) A balanced binary tree structure is imposed on the intervals, taking as leaves of the tree the intervals from earliest to latest, and having the nonleaf nodes represent new intervals that span the intervals of their children. The algorithm then sweeps up through the tree, computing best sets. For two consecutive time intervals I_1 and I_2, where I_1 precedes I_2, the set of jobs tentatively chosen to be scheduled in the spanning interval I is generated (roughly) as follows. This set will include the set of jobs tentatively chosen for I_1 plus a best set selected from jobs chosen from I_2 unioned with jobs not chosen from I_1.

This basic approach is fairly straightforward, but its correctness is not. Choosing best sets allows maximum flexibility in scheduling, since the jobs not chosen to be scheduled in the interval must be scheduled at a later time. But it is far from obvious that such best sets exist for any given time interval. We show in a lengthy proof by contradiction that best sets do in fact exist. The proof uses a nonobvious measure of the size of a problem. Furthermore, it is not obvious that the set of jobs not chosen in I_2 would also not be chosen in I. The correctness of this assertion depends on an involved proof by contradiction that is similar to the proof of the existence of a best set.

There is a crucial feature of our solution that we have not yet discussed. In order to be able to insert idle time into the schedule, each time interval mentioned above must actually represent a family of $O(n^2)$ time intervals, whose starting times differ by less than 1, and whose ending times differ by less than 1. In combining sets of chosen jobs for the families of intervals for I_1 and I_2, each interval in the family for I results from considering $O(n)$ combinations of individual intervals, one from I_1 and one from I_2. Some of the combinations do not necessarily result in a schedule. To test feasibility, we use what we call a "template." A template is formed using the set of jobs whose deadlines are in the given interval. From among all sets of these jobs that can be scheduled within the interval, the template is the set of deadlines that is smallest among such sets of jobs. We prove that such a template exists. The proof yields an elegant mirror-image approach to computing the set.

Our paper is organized as follows. In §2 we prove the existence of a canonical form for schedules, which is based on certain types of time intervals that we identify. In §3 we extend our characterizations to prove the existence of best sets and establish properties that lead to their fast computation. In §4 we present our NC algorithm in its entirety and analyze its performance.

2. Properties of intervals and schedules. In this section we identify a canonical form for schedules that we use in our parallel divide-and-conquer algorithm. The canonical form contains two types of time intervals induced by a set of jobs. In a "prime interval," certain jobs must be scheduled within the interval, and these jobs can always be tightly packed together. We identify a maximal set of prime intervals that are "compatible," called "cover intervals." The intervals that fall between consecutive cover intervals, called "gaps," are the more difficult to schedule. When no additional jobs can be scheduled in a gap, then the jobs can be tightly packed together in two groups, separated by a section of free space in which no job is scheduled. We show that for every schedule, there is a corresponding canonical schedule. This notion of a canonical schedule forms the basis of further characterizations in §3.

We first define some basic terms and establish some simple properties. Each job i has a release time r_i, a deadline d_i, and it should be processed for one unit of time in the interval $[r_i, d_i)$. The interval is closed on the left end to indicate that the job can start at r_i, and the interval is open on the right end to indicate that the job should be completed by d_i. Let s_i be the start time of job i and c_i be the completion time of job i, $c_i = s_i + 1$. The interval $[s_i, c_i)$ represents the time job i is processed. A *schedule* is an assignment of start times to jobs, such that the difference between the start times of any two jobs is at least one and for each job i,

$r_i \le s_i$ and $c_i \le d_i$. A set of jobs can be *scheduled in an interval* $[a, b)$ if and only if there is a schedule of the jobs such that for each job i, $s_i \ge a$ and $c_i \le b$.

Our approach is based on considering time intervals with certain interesting properties. We say that a job i is *contained* in an interval $[a, b)$ if and only if $a \le r_i$ and $d_i \le b$. We consider intervals $[a, b)$ such that $a = r_i$ for some job i contained in $[a, b)$ and $b = d_j$ for some job j contained in $[a, b)$. An interval $[a, b)$ has *looseness* x if and only if there are precisely $b - a - x$ jobs contained in it. If any interval has negative looseness, then no schedule is possible. We shall assume for the remainder of this section that all intervals have nonnegative looseness.

We are now ready to define an interval such that the jobs contained within it are easy to identify and easy to schedule. A *constrained interval* is an interval whose looseness is less than 1. A *prime interval* is a constrained interval $[a, b)$ such that there is no constrained interval $[a', b')$ properly contained in $[a, b)$. Figure 1(a) shows a set of jobs. Each job is represented by an interval in which the left endpoint is its release time and the right endpoint is its deadline. There are quite a few constrained intervals in Fig. 1(a). One constrained interval is [7.2, 11.1), which contains three jobs, 4, 5, and 6, and has looseness 0.9. It is not a prime interval because the constrained interval [8.0, 10.4) is contained in it. The prime intervals in Fig. 1(a) are [8.0, 10.4), [13.0, 15.6), and [13.6, 16.3).

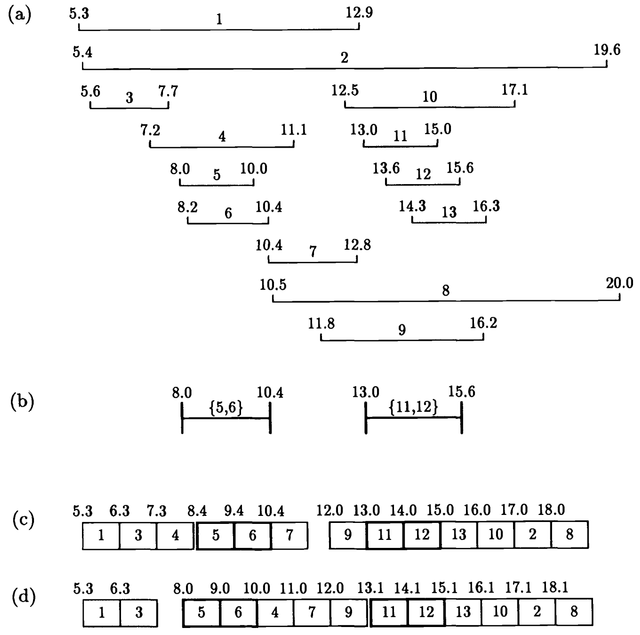


FIG. 1. (a) *A set of jobs with release times and deadlines*; (b) *A cover for this set of jobs*; (c) *A schedule for anchored gaps* [5.3, 8.4), [10.4, 13.0) *and* [15.0, 20.0); (d) *A schedule for anchored gaps* [5.3, 8.0), [10.0, 13.1), *and* [15.1, 20.0).

A prime interval is useful, because the jobs that must be scheduled within the interval can be packed tightly together with no free space between them and with a variable amount of free space on either end of the interval. We prove this in the following lemma.

LEMMA 2.1. *Let $[a, b)$ be a prime interval with looseness x. For any y, $0 \le y \le x$, the jobs contained in $[a, b)$ can be scheduled in the interval $[a + y, b - x + y)$.*

Proof. Suppose that the jobs contained in $[a, b)$ cannot be scheduled in the interval $[a + y, b - x + y)$ for some value y, $0 \leq y \leq x$. Without loss of generality, assume that $a + y$ is an integer. (Otherwise, we can subtract $a + y - \lfloor a + y \rfloor$ from all values to generate an equivalent problem.) For each job i contained in $[a, b)$, set $r_i' = \lceil r_i \rceil$, and $d_i' = \lfloor d_i \rfloor$. The resulting jobs are called *modified*. Since all r_i' and d_i' are integers, and there is no schedule for the modified jobs, then there must be an interval $[a', b')$ properly contained in $[a, b)$, with a' and b' integers, such that there are at least $b' - a' + 1$ modified jobs contained in $[a', b')$. Let a'' be the earliest release time of an original job whose corresponding modified job is contained in $[a', b')$, and let b'' be the latest deadline of an original job whose corresponding modified job is contained in $[a', b')$. Then there are at least $b' - a' + 1$ original jobs in the interval $[a'', b'')$, which is of length $b'' - a'' < (b' + 1) - (a' - 1) = b' - a' + 2$. Thus the interval $[a'', b'')$ has looseness less than 1, i.e., it is a constrained interval. Furthermore, interval $[a'', b'')$ is properly contained in $[a, b)$, for the following reason. Since $b' - a' + 1 \leq (b - x + y) - (a + y)$, and a', b', $a + y$ and $b - x + y$ are all integers, either $a' \geq a + y + 1$ or $b' \leq b - x + y - 1$. Thus either $a'' > a' - 1 \geq a + y$ or $b'' < b' + 1 \leq b - x + y$, and it follows that $[a'', b'')$ is properly contained in $[a, b)$. This is a contradiction to the original assumption that $[a, b)$ is a prime interval. It follows that the original jobs can be scheduled in $[a + y, b - x + y)$. □

Within isolated prime intervals, jobs are easy to schedule, but when these intervals overlap, a schedule in one interval affects the schedule in the other. Two prime intervals $[a, b)$ and $[a', b')$, with $a < a'$, are *compatible* if and only if $b - a' < 2$. If two prime intervals are compatible, then they do not contain a common job. This can be shown as follows. Suppose each interval contained job i. Then $a' \leq r_i < d_i \leq b$, which implies that $d_i - r_i < 2$, and thus $[r_i, d_i)$ would be a constrained interval contained inside an interval $[a, b)$ claimed to be prime. This is not possible.

Because of the incompatibility of certain prime intervals, we focus on a subset of the set of all prime intervals. A maximal set of prime intervals that are pairwise compatible is a *cover* for the set of jobs. A cover is shown in Fig. 1(b). Jobs 5 and 6 must be scheduled in [8.0, 10.4) (indicated by {5, 6} in the figure), and jobs 11 and 12 must be scheduled in [13.0, 15.6). Not all prime intervals are compatible. The prime interval [13.6, 16.3) is not part of the cover since [13.0, 15.6) and [13.6, 16.3) are not compatible. Both of these prime intervals contain job 12. For any two compatible prime intervals that overlap, the jobs in the prime interval with the smaller left endpoint are scheduled before the jobs in the other prime interval. For example, if job 4 in Fig. 1 had deadline 8.6, then there would be an additional prime interval in the cover, [7.2, 8.6), that contains job 4. The compatible prime intervals [7.2, 8.6) and [8.0, 10.4) overlap, so job 4 would have to be scheduled before jobs 5 and 6.

The precise scheduling of jobs contained in the prime intervals of a cover, called *cover intervals*, is dependent on the scheduling of the jobs not contained in those intervals. Given a cover, let $[a, b)$ and $[a', b')$ be two consecutive cover intervals. The *gap* between intervals $[a, b)$ and $[a', b')$ is the interval $[a'', b'')$, where $a'' = a + \lfloor b - a \rfloor$ and $b'' = b' - \lfloor b' - a' \rfloor$. The value a'' is the theoretically earliest possible time after a at which a job not contained in $[a, b)$ can be started in a schedule, and b'' is the latest possible time before b' at which a job not contained in $[a', b')$ can be completed. Note that if the looseness of $[a, b')$ is 0, then gap $[a'', b'')$ constitutes the empty interval. Of course, whether a job can actually start at a'' in a schedule depends on whether the jobs contained in $[a, b)$ are scheduled to complete by a''. It is important to maintain this flexibility in the definition of a gap. If two cover intervals overlap, then in any schedule the gap between them will either be empty or will contain precisely one job, which is not contained in either of the cover intervals. If $b'' - a'' < 1$, then in any schedule a job that is not contained in a cover interval will not be scheduled in the gap.

For uniformity, we require that each gap is always surrounded by two cover intervals. This is easily taken care of by adding a cover interval with looseness 0 at the beginning and end of the schedule. Let $r_{\min}$ be the minimum release time and $d_{\max}$ the maximum deadline in the problem. Two new jobs are introduced, job $n+1$ with $r_{n+1} = r_{\min} - 1$ and $d_{n+1} = r_{\min}$, and job $n+2$ with $r_{n+2} = d_{\max}$ and $d_{n+2} = d_{\max} + 1$. This forces two new cover intervals, $[r_{n+1}, d_{n+1})$ and $[r_{n+2}, d_{n+2})$, to be included in the cover. Clearly, the original n jobs can be scheduled if and only if the new set of $n+2$ jobs can be scheduled. In Fig. 1, two additional jobs would be added on the ends of the overall interval, job 14 with $r_{14} = 4.3$ and $d_{14} = 5.3$ and job 15 with $r_{15} = 20.0$ and $d_{15} = 21.0$. This would result in cover intervals on the ends of the overall interval; these are not shown.

We restrict our attention to specific subintervals of gaps. For a given cover, let $[a, b)$ and $[a', b')$ be two consecutive cover intervals with looseness x and x', respectively. An *anchored gap* is an interval $[a'', b'')$, where $b - x \leq a'' \leq b$, $a' \leq b'' \leq a' + x'$, a'' differs from some release time by an integer, and b'' differs from some release time by an integer. At least one anchored gap $[b - x, a' + x')$ for gap $[b, a')$ exists, since the looseness of $[a, b')$ is by assumption at least 0. If $x + x' = 0$ and $b = a'$, then we say that there is one anchored gap for gap $[b, a')$, namely $[b, a')$, which is empty. Since each of a'' and b'' can be one of at most n values, there are at most n^2 anchored gaps for any gap.

If there is a schedule of jobs in an anchored gap that almost fills the anchored gap, then there is a schedule in which the jobs are packed together in two groups with free space between the two groups. For a given schedule, a *hole* in an anchored gap $[a'', b'')$ is a nonempty interval $[a''', b''')$ contained in $[a'', b'')$, such that no jobs are scheduled in $[a''', b''')$ and both $a''' - 1$ and b''' are start times for jobs in the schedule.

LEMMA 2.2. *For any set of jobs that has a schedule S, and for any cover for the set of jobs, there is a schedule S' such that for each anchored gap $[a, b)$ that has $\lfloor b - a \rfloor$ jobs scheduled within it, there is at most one hole in the anchored gap.*

Proof. Let $[a, b)$ be an anchored gap in S that has $\lfloor b - a \rfloor$ jobs scheduled in it. We claim that there is a schedule of the jobs in the anchored gap such that there is only one hole within it. The proof of the claim is by induction on h, the number of jobs scheduled between first and last holes in $[a, b)$. For the basis, with $h = 0$, the claim is trivially satisfied. For the induction step, with $h > 0$, assume that the claim holds whenever there are fewer than h jobs scheduled between the first and last holes of $[a, b)$. Let $[a_1, b_1)$ be the first hole in $[a, b)$ and let $[a_l, b_l)$ be the last hole in $[a, b)$. We shall show that some of the jobs scheduled in $[b_1, a_l)$ can be scheduled starting at a_1 or finishing at b_l, thus reducing the number of jobs scheduled between the first and last holes.

For each job i scheduled in $[b_1, a_l)$, temporarily reset the release time r_i to be $r'_i = \max\{r_i, a_1\}$ and the deadline d_i to be $d'_i = \min\{d_i, b_l\}$. Since each such job i is scheduled in $[b_1, a_l)$, the schedule would still be valid if we had reset r_i to be $\max\{r_i, b_1\}$ and d_i to be $\min\{d_i, a_l\}$. Since $a_1 < b_1$ and $b_l > a_l$, the resetting we actually do is no more restrictive, and thus the schedule is still valid. Let $a' = \min\{r'_i\}$ and $b' = \max\{d'_i\}$. Since $[a, b)$ contains $\lfloor b - a \rfloor$ jobs, and $(b_1 - a_1) + (b_l - a_l) < 1$, we have $(b_1 - a') + (b' - a_l) < 1$. Thus $[a', b')$ is a constrained interval with respect to the modified release times and deadlines. Note that if both $a' > a_1$ and $b' < b_l$, then $[a', b')$ would be a constrained interval with respect to the original release times and deadlines, and would be compatible with all cover intervals, a contradiction to the cover being maximal.

Without loss of generality, assume $a' = a_1$. (The argument for $b' = b_l$ is similar.) Now identify the smallest value $b'' < b'$ such that $[a_1, b'')$ is a constrained interval with respect to the modified release times and deadlines. Any interval that is a constrained interval with respect to the modified release times and deadlines and is contained within $[a', b')$ must have either a' or b' as an endpoint. Thus $[a_1, b'')$ does not properly contain a constrained interval, and is thus a prime interval with respect to the modified release times and deadlines. The set of jobs whose scheduled positions in S overlap with $[a_1, b'')$ is precisely the set of jobs contained in $[a_1, b'')$ with respect to the modified release times and deadlines. By Lemma 2.1, this set of jobs can be scheduled in $[a_1, a_1 + \lfloor b'' - a_1 \rfloor)$. The remaining jobs are scheduled as they were in S. Thus the first hole will now begin at $a_1 + \lfloor b'' - a_1 \rfloor$ rather than a_1, and there will be $h - \lfloor b'' - a_1 \rfloor$ jobs between the first and last holes. By the induction hypothesis, the jobs in $[a, b)$ can be rescheduled to yield just one hole. This completes the proof of the claim.

The proof of the lemma follows by handling in turn each anchored gap $[a, b)$ that has $\lfloor b - a \rfloor$ jobs scheduled in it and has more than one hole. □

A schedule of the jobs for the anchored gaps [5.3, 8.4), [10.4, 13.0), and [15.0, 20.0) is shown in Fig. 1(c). A schedule of the jobs for anchored gaps [5.3, 8.0), [10.0, 13.1), and [15.1, 20.0) is shown in Fig. 1(d). Lemma 2.2 is illustrated by these schedules. In each anchored gap the jobs can be scheduled so that there is at most one hole within the gap. Note that there is no schedule when the first anchored gap is [5.3, 8.2). This follows since jobs 3 and 4 would have to be scheduled in the first anchored gap, and jobs 1, 7, 9, 11, 12, and 13 would have to be scheduled by 16.3, with 16.2 being the earliest at which they could all be finished. But then job 10 cannot start before 16.2, and thus cannot finish by its deadline.

We now show that if there is a schedule, then there is a schedule such that the jobs are nicely packed and the starting times of the jobs are convenient values. Given a set J of jobs, a *breakpoint* is any number x such that for some job j, $x - r_j$ is an integer. Given a cover, those jobs that must be scheduled within the cover intervals are called *cover jobs*, and those jobs that are scheduled within anchored gaps are called *gap jobs*. A *canonical schedule* is a schedule in which each job starts at a breakpoint, the cover jobs are scheduled tightly together within the cover intervals, and there is at most one hole in any anchored gap $[a, b)$ in which $\lfloor b - a \rfloor$ gap jobs are scheduled. We show in the next theorem that we can restrict our algorithm to finding a canonical schedule.

THEOREM 2.1 (canonical schedule). *For any set of jobs that has a schedule and for any cover for the set of jobs, there is a corresponding canonical schedule.*

Proof. Consider any schedule S and any cover for the set of jobs. Consider any cover interval $[a, b)$ with looseness x. The jobs contained in $[a, b)$ will be the only jobs completely scheduled in $[a, b)$, since $x < 1$. Let $a + y$ be the earliest start time of any of these jobs. By Lemma 2.1, the jobs contained in $[a, b)$ can be scheduled in $[a + y, b - x + y)$, which means that they are scheduled without any free space between them. It follows that there is a schedule S' such that for each cover interval, the cover jobs are scheduled without any free space between them within the cover interval. Given S', Lemma 2.2 establishes that there is a schedule S'' such that there is only one hole in each anchored gap $[a, b)$ that has $\lfloor b - a \rfloor$ jobs scheduled in it.

We derive a canonical schedule S''' from S'' that preserves the same order of the jobs, though it may shift their start times. For this discussion, consider there to be one additional hole, that starts when the last job completes. Consider the first hole $[a_1, b_1)$ in the schedule, and let J_1 be the set of jobs with start times before a_1. Let y_1 be the maximum value that can be subtracted from the start of each job in J_1 such that a schedule still remains. Subtract y_1 from each such start time. Clearly, some job in J_1 is starting at its release time, and all the rest start at a time that differs from this time by an integer. For each succeeding hole $[a_i, b_i)$, $i > 1$, let J_i be the set of jobs scheduled between this hole and the preceding one. Let y_i be the maximum value that can be subtracted from the start of each job in J_i such that a schedule still remains. Subtract y_i from each such start time. Clearly, either some job in J_i is starting at its release time, or the start time of the first job in J_i equals the completion time of the last job in J_{i-1}. In the latter case, each job starts at a time that differs from some release time by an integer, by transitivity. Thus each job in S''' starts at a time that differs from some job's release time by an integer. It follows that S''' is a canonical schedule. □

By Theorem 2.1, we can limit deadlines to being breakpoints. Thus we may assume as preprocessing that each deadline d_j is reset to the largest breakpoint no larger than d_j. Alternatively, breakpoints could be defined in terms of deadlines, and each release time r_j could be reset to the smallest breakpoint no smaller than r_j.

Consider the schedules in Figs. 1(c) and 1(d). In both schedules, the cover jobs are tightly scheduled together within the cover intervals, and there is at most one hole in any anchored gap. Since $r_1 = 5.3$, $r_7 = 10.4$, and $r_{11} = 13.0$, every job in the schedule in Fig. 1(c) starts at a breakpoint. Thus the schedule in Fig. 1(c) is canonical. However, since there is no job whose release time has fractional part .1, the schedule in Fig. 1(d) is not canonical.

3. Best r-sets, best d-sets, and templates.

The characterization of canonical schedules in the last section is not sufficient for designing a fast parallel algorithm. In particular, no method is implied to choose an appropriate set of jobs to be scheduled in a gap, and no method is identified for choosing suitable endpoints of a gap, in the case that its bracketing cover intervals have nonzero looseness. In this section we discuss the existence and computation of what we shall define as a "best r-set," a best choice of a subset of jobs to be scheduled in an interval. To make best r-sets unique, we shall transform problem instances so that all release times are distinct and all deadlines are distinct. A best r-set is easy to compute when the interval is a gap, but is more complicated to compute when the interval contains a collection of gaps and cover intervals. In the latter case, we first establish the existence of the best r-set, and then show how to select a subset of the jobs that will form the best r-set if there is a schedule. To identify suitable endpoints for a gap, all possible choices can be considered, with a test performed to determine if the selected jobs can be scheduled. We examine what we call a "modified mirror image problem," and identify a template of deadlines that represents the minimum set of deadlines that will result in a schedule. The template can be compared to a set of selected jobs to determine if the endpoints were suitable.

We first discuss a transformation that will give us uniqueness with respect to the best sets that we shall introduce shortly. Let a set of jobs be *simple* if all release times are distinct and all deadlines are distinct. Given a set of jobs for which a schedule exists, we can reset release times and deadlines so as to make the set of jobs simple. While either of the following operations applies, perform it. If $r_j = r_k$ and $d_j \leq d_k$ for jobs j and k, then reset r_k to be $r_j + 1$. If $r_j < r_k$ and $d_j = d_k$, then reset d_j to be $d_k - 1$. Clearly, performing these operations does not affect whether or not a schedule exists. We assume for the remainder of this section that the set of jobs in the problem instance has been transformed so as to be simple.

We first introduce the notion of feasibility with respect to an interval, and then we define what we call an r-set. Let J be a set of jobs, and $[a, b)$ an interval. Recall that a job is contained in an interval $[a, b)$ if $r_j \geq a$ and $d_j \leq b$. Let $J[a, b)$ be the subset of all jobs in J that are contained in $[a, b)$. A set J is $[a, b)$*-feasible* if there is a schedule for $J[a, b)$. If there is no schedule for $J[a, b)$, then there cannot be a schedule for J. We next consider a partition of J based on release times. Let $J_r[a, b)$ be the subset of all jobs j in J such that $r_j \in [a, b)$. Note that $J[a, b) \subseteq J_r[a, b)$. A set A is an *r-set* for $[a, b)$ with respect to J if and only if J is $[a, b)$-feasible and A is a subset of $J_r[a, b)$ such that $J[a, b) \subseteq A$ and the jobs in A can be scheduled in $[a, b)$. We choose the name r-set, where r denotes release time, to emphasize that jobs are partitioned by their release times.

Since some r-sets are better than others when constructing a schedule, we define the notion of a "best" r-set. Consider an interval $[a, b)$. Clearly, the jobs that are contained in $[a, b)$ must be scheduled in $[a, b)$. We would also like to schedule as many additional jobs as possible, and choose those jobs to be as preferable as possible. For those jobs with release times in $[a, b)$ but with deadlines greater than b, it is preferable to choose to schedule in $[a, b)$ those jobs with the smallest deadlines that can be scheduled. This strategy allows the jobs that are not chosen to have a better chance of being scheduled in a later time interval, since they have larger deadlines. For any set A of jobs, let $d(A)$ denote the (multi)set of deadlines of jobs in A. We define the partial order relation $\preceq$ on (multi)sets of values. If X and Y are two sets of values, then $X \preceq Y$ if and only if for $i = 1, 2, \ldots, \min\{|X|, |Y|\}$, $x_i \leq y_i$, where x_i is the ith smallest value in X and y_i is the ith smallest value in Y. Thus we prefer a set A of jobs over a set B of jobs with respect to their deadlines if and only if $d(A) \preceq d(B)$. If we deal with a set of jobs in which all deadlines are distinct, equality between equal-cardinality subsets A and B will hold if and only if they are identical. An r-set A for $[a, b)$ with respect to J is a *best r-set* for $[a, b)$ with respect to J if and only if for any other r-set B for $[a, b)$ with respect to J, $d(A) \preceq d(B)$ and $|A| \geq |B|$. If J is simple, then A is a unique best r-set.

We discuss examples of best r-sets, using the set of jobs in Fig. 1(a). There are 6 jobs whose release times lie in [5.3, 8.2], jobs 1, 2, 3, 4, 5, and 6. We do not consider jobs 5 and 6 since they obviously cannot be scheduled within this interval. Job 3 is contained in [5.3, 8.2), and thus must be in any r-set. Jobs 3 and 4 form the best r-set for [5.3, 8.2) as $d(\{3, 4\}) \preceq d(B)$, for B that is any of $\{3\}$, $\{1, 3\}$, and $\{2, 3\}$. Similarly, jobs 7 and 9 form the best r-set for [10.2, 13.0). The best r-set for [15.0, 20.0) is empty since there are no jobs with a release time in the interval.

We discuss the existence and computation of the best r-set for two types of intervals, the simpler interval that does not contain a constrained interval and the more complex interval that can contain constrained intervals. The first type corresponds to an anchored gap in our algorithm, and the second type corresponds to what we will call an anchored multiple gap, which we consider in the combining step of our algorithm.

We first discuss the existence and computation of the best r-set with respect to a set J of jobs for an interval $[a, b)$ that contains no constrained intervals. Clearly, J is $[a, b)$-feasible. Computing the best r-set for $[a, b)$ is straightforward. We define the *discrete earliest deadline rule* applied to the jobs in $J_r[a, b)$ as follows. For each job i, let r_i' be the smallest value no smaller than r_i such that $b - r_i'$ is an integer. Then apply the earliest deadline first rule using modified release times on the interval $[b - \lfloor b - a \rfloor, b)$. Using the earliest deadline first rule results in a set of jobs A such that $d(A) \preceq d(B)$ and $|A| \geq |B|$ for any subset B of $J_r[a, b)$ that can be scheduled in $[a, b)$.

LEMMA 3.1. *Let J be a set of jobs, and $[a, b)$ an interval, such that there is no constrained interval contained in $[a, b)$. The subset of $J_r[a, b)$ scheduled in $[a, b)$ by the discrete earliest deadline rule is a best r-set for $[a, b)$ with respect to J.*

Proof. Consider any subset A of $J_r[a, b)$ that can be scheduled in $[a, b)$. We generate a set A' of jobs with modified release times so that each differs from b by an integer. For each job $j \in A$, define job $j' \in A'$ such that $d_{j'} = d_j$ and $r_{j'}$ is the smallest value no smaller than r_j such that $b - r_{j'}$ is an integer. We show that A' can be scheduled in $[a, b)$. Suppose that no schedule of all jobs in A' is possible. Then there is an interval $[a', b')$ that contains more than $\lfloor b' - a' \rfloor$ jobs with respect to A'. Let $a'' = \min\{r_j | j \in A, j'$ is the corresponding job in A', $r_{j'} \geq a'$ and $d_j' \leq b'\}$. Since $r_j' - r_j < 1$ for each $j \in A$ and corresponding $j' \in A'$, $(a' - a'') < 1$. Thus there are at least $\lfloor b' - a' \rfloor + 1 = \lfloor b' - a'' - (a' - a'') \rfloor + 1$ $\geq \lfloor b' - a'' - 1 \rfloor + 1 = \lfloor b' - a'' \rfloor$ jobs contained in $[a'', b')$ with respect to A. Since $J_r[a, b)$ can be scheduled in $[a, b)$, there are at most $\lfloor b' - a'' \rfloor$ jobs contained in $[a'', b')$ with respect to A. Thus $[a'', b')$ is a constrained interval with respect to A, which is a contradiction to the assumption that $[a, b)$ does not contain a constrained interval. Thus it follows that all jobs in

A' can be scheduled. Thus modifying release times as in the discrete earliest-deadline-first rule does not eliminate any r-set A. Once all release times are modified, the problem is equivalent to a problem in which all release times and deadlines are integers. Then choosing the jobs with earliest-deadlines-first clearly produces a best r-set. □

We now discuss the existence of a best r-set for an interval that can contain constrained intervals. A *multiple gap* is an interval $[a, b)$ such that a is the left endpoint of a gap, b is the right endpoint of a different gap, and thus there is at least one cover interval within $[a, b)$. An *anchored multiple gap* is an interval $[a, b)$ such that a is the left endpoint of an anchored gap, b is the right endpoint of a different anchored gap, and thus there is at least one cover interval within $[a, b)$. Our algorithm will compute best r-sets, if they exist, for certain anchored multiple gaps. We now prove the existence of the best r-set for an unrestricted interval $[a, b)$ and an $[a, b)$-feasible set of jobs that is simple.

THEOREM 3.1 (best r-set). *Let $[a, b)$ be an interval, and J be an $[a, b)$-feasible set of jobs that is simple. Then there is a (unique) best r-set for $[a, b)$ with respect to J.*

Proof. The proof is by contradiction.

We define an *r-problem* P to consist of an interval $[a, b)$ and an $[a, b)$-feasible set of jobs J with distinct release times and distinct deadlines. Define a *P-breakpoint* to be either a or a breakpoint with respect to J. For any interval $[a, b)$, $a < b$, define $b \ominus a$, the *size* of $[a, b)$, to be the number of distinct P-breakpoints that lie in this interval. (We view the operation $\ominus$ as discretized subtraction, and use the symbol $\ominus$ as we would the minus sign. Thus we assume that the first operand is no smaller than the second.) Let the *size* of job i be the size of the interval $[r_i, d_i)$. Then the *size* of P is the size of interval $[a, b)$ plus the sum of the sizes of all jobs in J plus the sum of the product of sizes for all pairs of jobs in J:

$$size(P) = (b \ominus a) + \sum_{i \in J} (d_i \ominus r_i) + \sum_{i,j \in J,\ j \neq i} (d_i \ominus r_i) * (d_j \ominus r_j).$$

We consider an r-problem P, consisting of interval $[a, b)$ and set J of jobs, that is of smallest size among those r-problems that do not satisfy the theorem. Since P is of smallest size, $J_r[a, b) = J$. We shall eliminate all but two r-sets from consideration, and deduce various properties about schedules for these r-sets. We then use these properties to generate a contradiction.

Note that some job j in J has release time a, otherwise there would be a corresponding smaller r-problem P', consisting of interval $[a', b)$ and set of jobs J, where a' is the next P-breakpoint after a. P' is a smaller r-problem than P, so P' would have a best r-set A' in interval $[a', b)$. The set A' would be the best r-set for P in interval $[a, b)$, which is a contradiction to the assumption that there is no best r-set in P. Thus, there is some job j with $r_j = a$.

We restrict our attention to two r-sets in P as follows. Consider all r-sets in $[a, b)$ that have every schedule in $[a, b)$ starting at a. Then any such r-set must contain job j, and j is scheduled starting at a in such a schedule. We infer a smaller r-problem P' with interval $[a + 1, b)$ and set of jobs $J - \{j\}$. There is a best r-set A' for P'. Let $A_1 = A' \cup \{j\}$. A_1 is an r-set for P. We next consider all r-sets in $[a, b)$ that have some schedule in $[a, b)$ that does not start at a. Let a' be the next breakpoint after a. We infer a smaller r-problem P'' with interval $[a', b)$ and set of jobs J. (Note that job j will not be in any r-set for P''.) There is a best r-set A_2 for P''. This set A_2 is an r-set for P.

Any r-set A for P must satisfy either $d(A_1) \preceq d(A)$ and $|A_1| \geq |A|$, or $d(A_2) \preceq d(A)$ and $|A_2| \geq |A|$, since any schedule for A in $[a, b)$ either starts at a or it does not start at a. Thus we can confine our attention to the r-sets A_1 and A_2. Note that each job in J must appear in either A_1, A_2, or both. Otherwise we could remove such a job and get a smaller r-problem. The smaller r-problem would have a best r-set, which would then be the best r-set for P, a contradiction. By Theorem 2.1, canonical schedules exist for A_1 and A_2. Let S_1 be a canonical schedule for A_1 in $[a, b)$, and S_2 be a canonical schedule for A_2 in $[a', b)$. We note several properties of S_1 and S_2.

CLAIM 3.1. *Every job that is scheduled in both S_1 and S_2 starts at its release time in one of the schedules and completes at its deadline in the other schedule. If a job is scheduled in just one of S_1 and S_2, then the job starts at its release time in that schedule, and its deadline is greater than b.*

Proof of claim. Let k be a job in J that is scheduled in either S_1, S_2, or both. Let s_k be k's earliest starting time in the two schedules. Suppose $r_k < s_k$. We generate a new r-problem P' by resetting r_k to s_k. Note that resetting r_k does not generate any new breakpoints, since choosing S_1 and S_2 to be canonical schedules guarantees that s_k is a breakpoint. Also note that if this causes J to no longer be simple, we can just apply the appropriate operations to reset release times. P' is a smaller r-problem since the size of job k (and possibly some other jobs) has been reduced. Since P' is smaller than P, there is a best r-set A' in P'. Schedules S_1 and S_2 are both schedules in P'. Thus A_1 and A_2 are both r-sets in P', so $d(A') \preceq d(A_1)$ and $|A'| \geq |A_1|$, and $d(A') \preceq d(A_2)$ and $|A'| \geq |A_2|$. Thus A' is the best r-set for P, which is a contradiction to the assumption that there is no best r-set in P. Thus $r_k = s_k$ for all k in J.

Suppose that k is scheduled in both S_1 and S_2 and let c_k be k's latest completion time in the two schedules. Suppose $d_k > c_k$. We generate a smaller r-problem P' by resetting d_k to c_k. Again we perform any additional operations needed to keep J simple. Since P' is smaller than P, there is a best r-set A' in P'. Note that any job other than k whose deadline is reset must have its deadline be at most b, and is thus in every r-set for P and for P'. Since both A_1 and A_2 also contain any job whose deadline is reset, $d(A') \preceq d(A_1)$ for P' if and only if $d(A') \preceq d(A_1)$ for P, and similarly for A' and A_2. Also note that $|A'| \geq |A_1|$ and $|A'| \geq |A_2|$. Since for any r-set A for P, either $d(A_1) \preceq d(A)$ or $d(A_2) \preceq d(A)$, A' is a best r-set for P, which is a contradiction to the assumption that there is no best r-set in P. Thus $d_k = c_k$ for all k in J.

Suppose job k is scheduled in just one of S_1 and S_2. Since $J = J_r[a, b)$ and J is $[a, b)$-feasible, any job in J with deadline at most b must be scheduled in both S_1 and S_2. Since job k is not in both S_1 and S_2, $d_k > b$. This completes the proof of Claim 3.1.

In the remainder of the proof, if resetting a release time or deadline causes J to no longer be simple, we apply the appropriate operations, as in the proof of the last claim, to make J simple once again. We note that whenever a deadline is changed, the job will appear in every corresponding r-set. Thus comparisons between the deadlines of r-sets using $\preceq$ are not affected.

For schedule S_p, where $p = 1$ or $p = 2$, and for any job h in J that is scheduled in S_p, let s_h^p be the start time of h in S_p and c_h^p be the completion time of h in S_p. We say that the scheduled position of a job g in S_1 *overlaps* that of job h in S_2 if and only if $\max\{s_g^1, s_h^2\} < \min\{c_g^1, c_h^2\}$.

CLAIM 3.2. *For any job that is scheduled in both S_1 and S_2, its scheduled positions in the two schedules overlap.*

Proof of claim. Suppose the claim is false. Let k be the job with the smallest deadline that is scheduled in both S_1 and S_2 such that its scheduled positions in the two schedules do not overlap.

We introduce five transformations for generating smaller r-problems that result in contradictions. The first is called an *exchange-d* transformation. Let g and h be jobs in J with $r_h < r_g$, $d_h < d_g$, and $s_h^p < s_g^p \leq s_h^{3-p}$, where $p = 1$ or $p = 2$. By Claim 3.1, it follows that $s_h^p = r_h$, $s_g^p = r_g$, $c_h^{3-p} = d_h$, and $c_g^{3-p} = d_g$. We transform r-problem P into a new r-problem P' by replacing jobs h and g with jobs h' and g', where $r_{h'} = r_h$, $d_{h'} = d_g$, $r_{g'} = r_g$, and $d_{g'} = d_h$. We call this transformation an exchange-d, because we exchange the deadlines for h and g. Schedules S_{3-p} and S_p for P are easily transformed into corresponding schedules

S'_{3-p} and $S_p{}'$ for P' in the following way. Schedule S'_{3-p} will be the same as S_{3-p}, except that g' will be in the position of h, and if g is in S_{3-p} then h' will be in the position of g. Schedule S'_p will be the same as S_p, except that h' will be in the position of h, and g' in the position of g. Thus the sets A'_{3-p} and A'_p corresponding to A_{3-p} and A_p are r-sets in P'. Figure 2 illustrates the exchange-d transformation of a problem P (which has schedules S_1 and S_2) into a problem P' (which has schedules S'_1 and S'_2). The release time-deadline intervals for the relevant jobs are also shown.

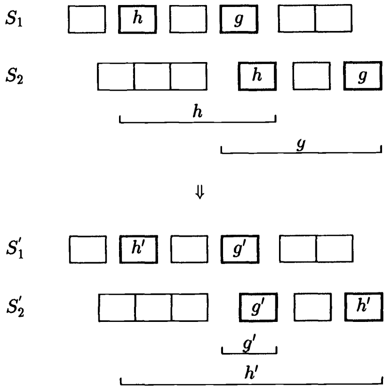


FIG. 2. *An exchange-d transformation.*

A schedule S' in P' can be transformed to a schedule S in P in the following way. Any job in S' that is not h' or g' is scheduled in the same position in S. Note that g' must be scheduled in S', and h' may or may not be scheduled in S'. If h' is scheduled overlapping in $[r_{h'}, r_{g'})$, then h is scheduled in S in the position of h' in S' and g is scheduled in S in the position of g' in S'. Otherwise, h is scheduled in S in the position of g' in S', and if h' is scheduled in S', then g is scheduled in S in the position of h' in S'. Any set A in P whose corresponding set A' in P' is an r-set in P' is itself an r-set in P. The size of P' is smaller than the size of P since

$$\begin{aligned}(d_g \ominus r_h) * (d_h \ominus r_g) &< ((d_g \ominus r_h) * (d_h \ominus r_g)) + ((d_g \ominus d_h) * (r_g \ominus r_h))\\ &= ((d_g \ominus r_h) * (d_h \ominus r_g)) + (d_g \ominus d_h) * ((d_g \ominus r_h) - (d_g \ominus d_h) - (d_h \ominus r_g))\\ &= ((d_g \ominus r_h) - (d_g \ominus d_h)) * ((d_h \ominus r_g) + (d_g \ominus d_h))\\ &= (d_h \ominus r_h) * (d_g \ominus r_g).\end{aligned}$$

Thus P' has a best r-set A', which is at least as large as any r-set of P. For set B as any of A_{3-p}, A_p or A, h is in B if and only if g' is in B', and g is in B if and only if h' is in B'. Thus for set B as any of A_{3-p}, A_p or $A, d(B) \preceq d(B')$ and $d(B') \preceq d(B)$. Since $d(A') \preceq d(A'_{3-p})$ and $d(A') \preceq d(A'_p)$, we have $d(A) \preceq d(A_{3-p})$ and $d(A) \preceq d(A_p)$, and thus A is a best r-set for P, which is a contradiction to the assumption that P does not have a best r-set. Thus whenever an exchange-d transformation can be applied, a contradiction can be achieved. This concludes the discussion of an exchange-d transformation.

The second transformation is called a *compress* transformation. Let g and h be jobs in J with $c_g^p = s_h^p$, $c_g^{3-p} = s_h^{3-p}$, and $s_h^p \leq s_h^{3-p} < c_h^p$, where $p = 1$ or $p = 2$. By Claim 3.1, it follows that $s_g^p = r_g$, $s_h^p = r_h$, $c_g^{3-p} = d_g$, and $c_h^{3-p} = d_h$. We generate a new r-problem P' in which the jobs g and h are compressed into one job h' such that $r_{h'} = r_h$ and $d_{h'} = d_h$. For any job i with $r_i < r_g$, reset r_i to $r_i + 1$. For any job i with $d_i < d_g$, reset d_i to $d_i + 1$. Reset a to $a + 1$. Note that the sets A'_{3-p} and A'_p in P' that correspond to A_{3-p} and A_p in P are r-sets in P'. The new r-problem P' is smaller than P since there is one fewer job, $(b \ominus a)$ is smaller, and $(d_i \ominus r_i)$ is no larger for any remaining job i. Thus P' has a best r-set A'. Let $A = A' \cup \{g, h\} - \{h'\}$. Then A is an r-set for P and is of maximum size among r-sets for P. Note that any job other than g or h that has its deadline changed must be in every r-set for P' and in every r-set for P. By Claim 3.1, $d_g < b$. It follows that $d(A) \preceq d(A_{3-p})$ and $d(A) \preceq d(A_p)$. Thus A is a best r-set for P, which is a contradiction to our initial assumption. Thus whenever a compress transformation can be applied, a contradiction can be achieved. Figure 3 illustrates the compress transformation.

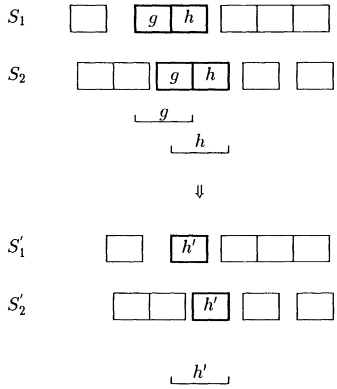


FIG. 3. *A compress transformation.*

The third transformation is a variation of the compress, called an *inverted compress*. Let g and h be jobs in J with $c_g^p = s_h^p$, $c_h^{3-p} = s_g^{3-p}$, and $s_g^p \leq s_h^{3-p} < c_g^p$, where $p = 1$ or $p = 2$. We generate a new r-problem P' in which the jobs g and h are compressed into one job h' such that $r_{h'} = s_h^p$ and $d_{h'} = d_g$. For any job i with $r_i < r_g$, reset r_i to $r_i + 1$. For any job i with $d_i < c_h^{3-p}$, reset d_i to $d_i + 1$. Reset a to $a + 1$. In a fashion analogous to that of the compress transformation, a contradiction can be achieved whenever an inverted compress transformation can be applied. Figure 4 illustrates the inverted compress transformation.

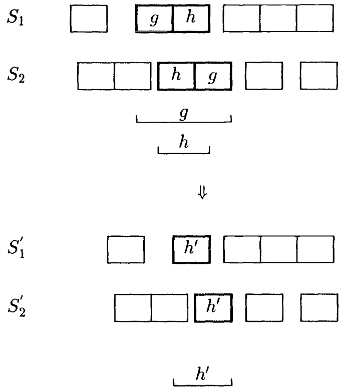


FIG. 4. *The inverted compress transformation.*

The fourth transformation is called an *increase-r* transformation. Let g and h be jobs in J with g scheduled before h in S_p, $c_g^p < s_h^p$, no other job scheduled between g and h in S_p, and $c_g^p < d_g$, where $p = 1$ or $p = 2$. By Claim 3.1, $r_g = s_g^p$. We reset r_g to be the next larger breakpoint, giving a smaller r-problem P', for which there would be a best r-set. This set would also be a best r-set for P, contradicting our initial assumption about P. Thus whenever an increase-r transformation can be applied, a contradiction can be achieved.

The fifth transformation is called a *decrease-d* transformation. Let g and h be jobs in J with g scheduled before h in S_p, $c_g^p < s_h^p$, no other job scheduled between g and h in S_p, and $r_h < s_h^p$, where $p = 1$ or $p = 2$. By Claim 3.1, $d_h = c_h^p$. We reset d_h to be the next smaller breakpoint, giving a smaller r-problem P', for which there would be a best r-set. Since job h must be in any r-set for P, changing its deadline does not affect whether or not an r-set is the best r-set for P. Hence the best r-set for P' would also be the best r-set for P, contradicting our initial assumption about P. Thus whenever a decrease-d transformation can be applied, a contradiction can be achieved.

We now proceed with a case analysis. Assume that $c_k^2 \le s_k^1$. (The argument for $c_k^1 \le s_k^2$ is essentially the same.) By Claim 3.1, $r_k = s_k^2$ and $d_k = c_k^1$. Suppose as Assumption (a1) that there is no job scheduled in S_2 during any part of the interval $[s_k^1, c_k^1)$. This case is illustrated in Fig. 5(a). Only the relevant jobs are shown. Then we can reset r_k to be s_k^1 and get a smaller r-problem P', for which A_2 can still be scheduled, with job k in interval $[s_k^1, c_k^1)$. This would mean that there would be a best r-set for P'. This set would also be the best r-set for P, contradicting the assumption about P. Thus (a1) cannot hold, and there is a job h that is scheduled in S_2 during some part of the interval $[s_k^1, c_k^1)$.

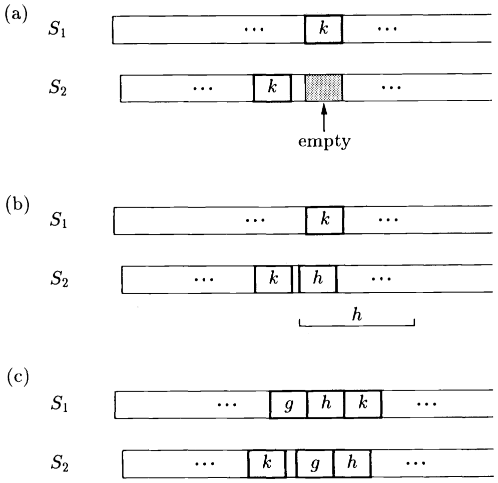


FIG. 5. *Selected cases in the proof of Claim* 3.2. (a) *Assumption* (a1); (b) *Assumptions* (a2) *and* (a2.1); (c) *Assumptions* (a2), (a2.2), *and* (a2.2.1).

Either $s_h^2 \le s_k^1 < c_h^2 \le c_k^1$, or both $s_k^1 < s_h^2 < c_k^1 < c_h^2$ and there is no job i such that $s_i^2 \le s_k^1 < c_i^2$. Suppose as (a2) that $s_h^2 \le s_k^1 < c_h^2 \le c_k^1$. By Claim 3.1 either $r_h = s_h^2$ or $d_h = c_h^2$. Suppose as (a2.1) that $r_h = s_h^2$. This case is illustrated in Fig. 5(b). Then an exchange-d transformation can be applied to jobs h and k. Thus (a2.1) does not hold, and $r_h \ne s_h^2$.

Thus we have that $d_h = c_h^2$. Since deadlines are distinct, $d_h < d_k$. Since k is the job with the smallest deadline whose positions in S_1 and S_2 do not overlap, job h is scheduled in positions in S_1 and S_2 that overlap. Thus job h is the job that precedes k in S_1. Furthermore, $c_h^1 = s_k^1$, since otherwise we could apply an increase-r transformation. Let g be the job that precedes job h in S_2. We have that $c_g^2 = s_h^2$, since otherwise we could apply a decrease-d transformation. Either $g \ne k$ or $g = k$. Suppose as (a2.2) that $g \ne k$. Either $d_g = c_g^2$ or $r_g = s_g^2$. Suppose as (a2.2.1) that $d_g = c_g^2$. This case is illustrated in Fig. 5(c). By choice of job k, the positions of g in S_1 and S_2 overlap. Thus job g is the job that precedes h in S_1. Furthermore, $c_g^1 = s_h^1$, since otherwise we could apply an increase-r transformation. A compress transformation can now be applied to jobs g and h. Thus (a2.2.1) does not hold, and $r_g = s_g^2$. Then $s_g^1 \geq c_k^1$ and $r_k < r_g$. An exchange-d operation can now be applied to jobs k and g. Thus (a2.2) does not hold, so $g = k$. Since $g = k$, we can apply an inverted compress transformation for k and h, which leads to a contradiction.

Thus (a2) does not hold, which means that there is a job h such that $s_k^1 < s_h^2 < c_k^1 < c_h^2$ and there is no job i such that $s_i^2 \leq s_k^1 < c_i^2$. Let job m be the job that precedes job h in S_2. Since there is no job i such that $s_i^2 \leq s_k^1 < c_i^2$, $c_m^2 < s_h^2$. We have that $r_h = s_h^2$, since otherwise we could apply a decrease-d transformation on jobs m and h. Let job l be the job that precedes job k in S_1. Note that $c_l^1 = s_k^1$, since otherwise we could apply a decrease-d transformation on jobs l and k. Also, $d_l < d_k$, since otherwise we could apply an exchange-d transformation for k and l. By choice of k, the scheduled positions of l in S_1 and S_2 overlap, and since there is no job i such that $s_i^2 \leq s_k^1 < c_i^2$, it follows that $m = l$. Since there is no job i such that $s_i^2 \leq s_k^1 < c_i^2$, we have $c_l^2 \leq s_k^1$, which means that $c_l^2 \leq c_l^1$, from which it follows that $d_l = c_l^1$. Then $c_l^2 = c_l^1$, since otherwise we could apply an increase-r transformation to l and h. Since $c_l^1 = d_l$ and $c_l^2 = c_l^1$, it follows that $s_l^2 = s_l^1$.

Let g be the job that precedes job l in S_2. Either $g = k$ or $g \neq k$. Suppose as (a3) that $g = k$. This case is illustrated in Fig. 6(a). We must have $c_g^2 = s_l^2$, since otherwise we could apply an increase-r transformation for g and l. We can then apply an inverted compress transformation for k and l. This transformation leads to a contradiction. Thus (a3) does not hold, and $g \neq k$. Let f be the job that immediately precedes job l in S_1. Either $f = g$ or $f \neq g$. Suppose as (a4) that $f = g$. This case is illustrated in Fig. 6(b). It follows that $r_g = d_g - 1$, by an argument similar to the one that showed that $r_l = d_l - 1$. We can then apply a modified compress transformation to g and l, with the only difference being the following. For any i with $r_i < r_g$, reset r_i to $r_i + r_l - r_g$. For any i with $d_i < d_l$, reset d_i to $d_i + r_l - r_g$. This leads to a contradiction.

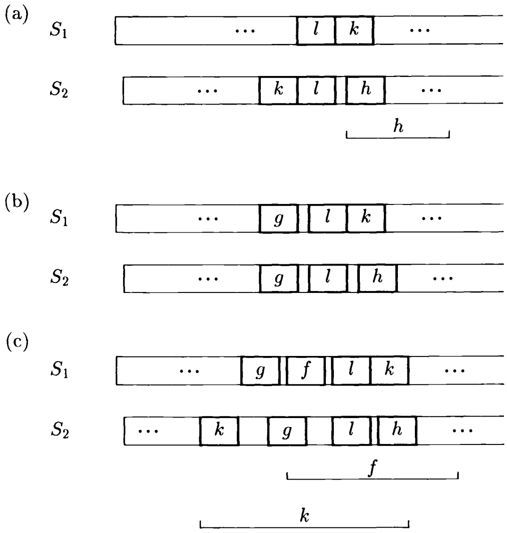


FIG. 6. *Selected cases in the proof of Claim* 3.2. (a) *Assumption* (a3); (b) *Assumption* (a4); (c) *Final contradiction.*

Thus (a4) does not hold, and $f \neq g$. Either $s_g^2 = r_g$ or $c_g^2 = d_g$. Suppose $s_g^2 = r_g$. Then we can perform an exchange-d operation on jobs g and k, which leads to a contradiction.

Thus $c_g^2 = d_g$. By choice of k the positions of g in S_1 and S_2 overlap, which means that the positions of f in S_1 and S_2 cannot overlap. This case is illustrated in Fig. 6(c). Thus $s_f^1 = r_f$ and $d_f > d_k$. But then we can reset d_k to be c_f^1 and r_f to be s_k^1. This gives a smaller r-problem P', which will thus have a best r-set. It follows that this set will also be a best r-set for P, a contradiction. At this point, all cases have been exhausted. Thus there can be no job k whose scheduled positions in S_1 and S_2 do not overlap. This completes the proof of Claim 3.2.

We are now ready to generate the contradiction to the assumption that the theorem does not hold. Let j be the job scheduled at a in S_1 and let h be the first job scheduled in S_2. If $j = h$, then we generate a new r-problem P' by deleting h and resetting a to $a + 1$. Then P' is smaller than P and thus it has a best r-set A'. Let $A = A' \cup \{h\}$. Then $d(A) \preceq d(A_1)$ and $|A| \geq |A_1|$, and $d(A) \preceq d(A_2)$ and $|A| \geq |A_2|$, which yields a contradiction, so $j \neq h$. Since no other job in S_2 can overlap j in S_1, job j is not in S_2, by Claim 3.2. Either $s_h^2 \geq c_j^1$ or $s_h^2 < c_j^1$. If $s_h^2 \geq c_j^1$, then $A_2 \cup \{j\}$ can be scheduled in $[a, b)$, since j can be scheduled in $[a, a + 1)$ and A_2 can be scheduled in $[a + 1, b)$. But this would contradict A_2 being an r-set for P. Thus $s_h^2 < c_j^1$. It follows that $s_h^2 = r_h$. Either h is scheduled in S_1 or it isn't. If h is not scheduled in S_1, then $d_h > b$. We generate a new r-problem P' by removing j and h and resetting a to $a + 1$. In a similar manner to the argument above, this yields a contradiction. If h is scheduled in S_1, then $s_h^1 < c_h^2$. Since $d_j > b$, we generate a new r-problem P' by removing h and resetting r_j to $r_j + 1$ and a to $a + 1$. In a similar manner to the argument above, this yields a contradiction. At this point we have exhausted all cases. The theorem that there is a best r-set for $[a, b)$ then follows . □

COROLLARY 3.1.1. *Let $[a, b)$ be an interval, and J be an $[a, b)$-feasible set of jobs with distinct deadlines. Then there is a (unique) best r-set for $[a, b)$ with respect to J.*

Proof. If deadlines are distinct, then there do not exist two different subsets A and B of jobs such that $d(A) \preceq d(B)$ and $d(B) \preceq d(A)$. If there are any jobs in J with the same release time, do the following. For the subset of jobs in J with release times less than a, arbitrarily reset all release times to be distinct values less than a. This cannot change any r-set for $[a, b)$. Next, perform the appropriate operations to reset release times until all remaining release times are distinct. Note that a best r-set for $[a, b)$ will remain a best r-set for $[a, b)$. By Theorem 3.1 there will be a unique best r-set for $[a, b)$ in the transformed problem. Thus there will be a unique best r-set for $[a, b)$ in the original problem. □

We consider the problem of determining a best r-set for an anchored multiple gap, given the best r-sets for two adjacent anchored multiple gaps that span it. We first consider the simpler problem of recomputing the best r-set for an interval $[a, b)$ when one additional job with release time a is inserted into the set of jobs. We show that if the best r-set for $[a, b)$ changes at all, then the only change is that the new job replaces one of the jobs in the best r-set.

LEMMA 3.2. *Let $[a, b)$ be an interval, and J and $J' = J \cup \{j'\}$ be $[a, b)$-feasible sets of jobs with distinct deadlines, where j' is a job not in J with $r_{j'} = a$. Let A_1 be the best r-set with respect to J, and A_2 the best r-set with respect to J'. Then $A_2 \subseteq A_1 \cup \{j'\}$.*

Proof. The proof is by contradiction and is similar in structure to the proof of Theorem 3.1. We first note that by Corollary 3.1.1, best r-sets A_1 and A_2 exist.

We define an *r^+-problem* P to consist of an interval $[a, b)$, a set of jobs J, and an additional job $j' \notin J$ with $r_{j'} = a$ such that $J' = J \cup \{j'\}$ is $[a, b)$-feasible. We define P-breakpoint, size of an interval, and size of a job as in the proof of Theorem 3.1. The size of r$^+$-problem P is the size of interval $[a, b)$ plus the sum of the sizes of all jobs in J' plus the sum of the product of sizes of all pairs of jobs in J'.

We consider an r$^+$-problem P, consisting of interval $[a, b)$, set J of jobs, and additional job j', that is of smallest size among those r$^+$-problems that do not satisfy the lemma. Since

P is of smallest size, $J_r[a, b) = J$ and $J'_r[a, b) = J'$. Suppose that $A_2 \not\subseteq A_1 \cup \{j'\}$. Clearly, $j' \in A_2$, since otherwise $A_2 = A_1$.

Let S_1 be any schedule for A_1, and S_2 be any schedule for A_2. We note several properties of S_1 and S_2.

CLAIM 3.3. *Every job that is scheduled in both S_1 and S_2 starts at its release time in one of the schedules and completes at its deadline in the other schedule. Job j' completes in S_2 at its deadline. If a job in J is scheduled in just one of S_1 and S_2, then the job starts at its release time in that schedule, and its deadline is greater than b.*

Proof of claim. Let j be a job in J that is scheduled in either S_1, S_2, or both. Let s_j be j's earliest starting time in the two schedules. Suppose $r_j < s_j$. We generate a new r$^+$-problem P' by resetting r_j to s_j. Clearly A_1 is the best r-set in P' with respect to J and A_2 is the best r-set in P' with respect to J'. P' is a smaller r$^+$-problem since the size of job j has been reduced. Since P' is smaller than P, $A_2 \subseteq A_1 \cup \{j'\}$, which contradicts the assumption that P does not satisfy the lemma. Thus $r_j = s_j$.

Suppose that job j is scheduled in both S_1 and S_2 and let c_j be j's latest completion time in the two schedules. Suppose $d_j > c_j$. We generate a new r$^+$-problem P' by resetting d_j to c_j. If this causes two jobs to have the same deadline, then apply the appropriate operation to reset deadlines, as was discussed in the proof of Theorem 3.1. Any job whose deadline is changed must have its deadline be at most b, and thus will be in every r-set for P' with respect to J and with respect to J', and similarly for P. Thus A_1 and A_2 remain best r-sets for P' with respect to J and J', respectively. Since P' is smaller, it follows that $A_2 \subseteq A_1 \cup \{j'\}$, which gives a contradiction. Thus $d_j = c_j$.

If job j' completes in S_2 before its deadline, then $d_{j'}$ can be reset to $c_{j'}$. Note that this does not affect the comparisons for best r-set A_2, since j' is in A_2. This once again gives a smaller r$^+$-problem, yielding a contradiction.

Suppose job $j \in J$ is scheduled in one of S_1 and S_2. Since $J = J_r[a, b)$ and J is $[a, b)$-feasible, any j in J with deadline at most b must be scheduled in both S_1 and S_2. Since job j is not in both S_1 and S_2, $d_j > b$. This completes the proof of the claim.

CLAIM 3.4. *For any job in J that is scheduled in both S_1 and S_2, their scheduled positions in the two schedules overlap.*

Proof of claim. The proof of this claim is similar to the proof of Claim 3.2 in Theorem 3.1, but in this lemma we are addressing r$^+$-problems and derive contradictions to the assumption of this lemma. The proof of this claim is a straightforward transformation of the proof of Claim 3.2, and is omitted.

We are now ready to generate the contradiction to the assumption that the lemma does not hold. Suppose j' is not the last job scheduled in S_2. Let h be the last job scheduled in S_2 and k be the last job scheduled in S_1. If $k = h$, then we can generate a smaller r$^+$-problem by deleting this job and subtracting 1 from b, and thus achieve a contradiction. Thus, $k \neq h$. Suppose $s_k^1 \geq s_h^2$. Then $s_k^1 = r_k$ and $d_k > b$. If $d_h > b$, then replace h by k in S_2, and generate a smaller r$^+$-problem, since there is one less job, and achieve a contradiction. Thus $d_h \leq b$, which means that the job scheduled before k in S_1 is h. But then we can generate a smaller r$^+$-problem by removing h, resetting the release time of k to $r_k - 1$, and resetting the interval to $[a, b - 1)$. This again leads to a contradiction. The argument is similar if $s_k^1 < s_h^2$.

Thus j' is the last job scheduled in S_2. Let h be the first job scheduled in S_2 and k be the first job scheduled in S_1. If $k = h$, then we can generate a smaller r$^+$-problem, which gives a contradiction, so $k \neq h$. Suppose $s_h^2 \leq s_k^1$. Then $s_h^2 = r_h$ and $d_h > b$. Then h and j' can be swapped in S_2 and the r_h reset to $s_{j'}^2$ before the swap. The size of a job has been reduced, resulting in a smaller r$^+$-problem, which gives a contradiction, so $s_h^2 > s_k^1$. Then, $d_k > b$. If $d_h > b$, then again, h and j' can be swapped in S_2, generating a smaller r$^+$-problem, which leads to a contradiction. Thus, $d_h \leq b$, which means that the second job scheduled in S_1 is h. But this leads to a smaller r$^+$-problem. This achieves the final contradiction, as we have shown that $s_h^2 \leq s_k^1$, $s_h^2 \geq s_k^1$, and $s_h^2 \neq s_k^1$. Thus, all jobs except $j' \in S_2$ appear in S_1. □

We now consider recomputing the best r-set when a set of additional jobs is introduced. We show that a job that is not in the original best r-set cannot appear in the recomputed best r-set.

THEOREM 3.2. *Let $[a, b)$ be an interval, and J' and $J \subset J'$ be $[a, b)$-feasible sets of jobs with distinct deadlines, and where each job $j' \in J' - J$ has $r_{j'} = a$. Let A_1 be the best r-set with respect to J, and let A_2 be the best r-set with respect to J'. Then $A_2 \subseteq A_1 \cup (J' - J)$.*

Proof. We first note that by Corollary 3.1.1, best r-sets A_1 and A_2 exist. Our proof is by induction on $|J' - J|$. For the basis, we have $|J' - J| = 1$. The basis case holds by Lemma 3.2. For the induction step, we have $|J' - J| > 1$. Assume as the induction hypothesis that the theorem holds for all values smaller than $|J' - J|$. Let j' be a job in $J' - J$. Let A_1 be the best r-set for $[a, b)$ with respect to J. By the induction hypothesis, the best r-set A_2 for $[a, b)$ with respect to $J' - \{j'\}$ satisfies $A_2 \subseteq A_1 \cup ((J' - J) - \{j'\})$. By Lemma 3.2, the best r-set A_3 for $[a, b)$ with respect to J' satisfies $A_3 \subseteq A_2 \cup \{j'\}$, which implies $A_3 \subseteq A_1 \cup (J' - J)$. □

In the last part of this section we concentrate on computing the best r-sets for intervals that contain cover intervals. As shown in Lemma 3.1, an interval $[a, b)$ that has no constrained interval contained within it (and hence no cover interval contained within it) is $[a, b)$-feasible and the corresponding schedule can be computed easily using the discrete earliest deadline rule. Determining $[a, b)$-feasibility is more involved when the interval contains cover intervals because we do not know in advance when the first job of a cover interval should be started. For certain choices of the starting time of the first job in a cover interval, there may be no schedule. Our approach is to consider all possible starting times of an appropriate cover interval, and compute a set of jobs that is the best r-set if the set is $[a, b)$-feasible. Then the $[a, b)$-feasibility of the set can be tested by comparing the set of deadlines for a certain subset of its jobs with a "template" generated from a "mirror-image problem." The template is composed of the smallest allowable deadlines for jobs that can be scheduled in an appropriate portion of $[a, b)$.

First we define the notions of d-set, best d-set, and template. Let $[a, b)$ be an interval, J a set of jobs, and $J_d[a, b)$ be the subset of all jobs j in J such that $d_j \in (a, b]$. The set A is a *d-set* for $[a, b)$ with respect to J if and only if J is $[a, b)$-feasible and A is a subset of $J_d[a, b)$ that includes all jobs in $J[a, b)$, and the jobs in A can be scheduled in $[a, b)$. We choose the name d-set, where d denotes deadline, to stress the partitioning of jobs by their deadlines. A d-set A for $[a, b)$ with respect to J is a *best d-set* for $[a, b)$ with respect to J if for any other d-set B for $[a, b)$ with respect to J, $d(A) \preceq d(B)$ and $|A| \geq |B|$. The set of deadlines of the best d-set for $[a, b)$ with respect to J is called a *template* for $[a, b)$ with respect to J.

We discuss examples of templates using the set of jobs in Fig. 1(a). There are six jobs whose deadlines lie within (15.0, 20.0], jobs 2, 8, 9, 10, 12, and 13. Job 12 is not considered further since it cannot be scheduled within the interval. The best d-set for [15.0, 20.0) consists of the four jobs 2, 8, 9, and 10. Thus the template for [15.0, 20.0) is the set {16.2, 17.1, 19.6, 20.0}, consisting of the deadlines of jobs 2, 8, 9, and 10. The deadline of job 13 is not in the template since only one of jobs 9 and 13 can be scheduled in [15.0, 16.4), and job 13 has a later deadline. Similarly, the template for [5.3, 8.2) consists of the deadline of job 3, and the template for [10.2, 13.0) consists of the deadlines of jobs 7 and 1.

A template can be used to test for feasibility as follows. Let $[a, b)$ and $[a', b')$ be two consecutive anchored multiple gaps, where $a' - b$ equals the number of jobs in the cover interval separating them. Suppose that J is $[a, b)$-feasible, $[a', b')$-feasible, and $[b, a')$-feasible. We want to determine if there is a schedule of jobs contained in $[a, b')$ that has $a' - b$ cover jobs scheduled in $[b, a')$. (This is a constrained type of $[a, b')$-feasibility.) Let B be all jobs that are contained in $[a, b')$ except those in the best r-set for $[a, b)$ and those to be scheduled in $[b, a')$. Let A be the best d-set for $[a', b')$ with respect to J. Then there is a desired schedule if and only if $d(A) \preceq d(B)$ and $|B| \leq |A|$. A proof of this claim can be sketched as follows. Clearly, there is a schedule of jobs contained in $[a, b')$ that has $a' - b$ cover jobs scheduled in $[b, a')$ if and only if B can be scheduled in $[a', b')$. Suppose B can be scheduled in $[a', b')$. Then B is a d-set for $[a', b')$. Since A is a best d-set, $d(A) \preceq d(B)$ and $|B| \leq |A|$. Suppose $d(A) \preceq d(B)$ and $|B| \leq |A|$. Since A is a d-set, every job contained in $[a', b')$ is a member of A. By definition, every job contained in $[a', b')$ is a member of B. All remaining jobs in either A or B have a release time earlier than a'. Thus given a schedule for A, a schedule for B can be constructed as follows. Any job contained in $[a', b')$ is in the same position in the schedule for B as in the schedule for A. Let A' and B' be the remaining sets of jobs. Since $|B| \leq |A|$, we have $|B'| \leq |A'|$. Schedule the job in B' with the ith largest deadline in the position that the job in A' with the ith largest deadline was scheduled. Thus B can be scheduled in $[a', b')$.

We illustrate how a template is used. Consider the set of jobs in Fig. 1(a), but with $r_9 = 12.1$. The best r-set for anchored multiple gap $[5.3, 13.0)$ will have jobs 1 and 3 scheduled in the gap $[5.3, 8.0)$, and jobs 4, 7, and 2 in gap $[10.0, 13.0)$. It is not possible to schedule the remaining jobs 8, 9, 10, and 13 in $[15.0, 20.0)$. Job 13, with second smallest deadline, has a deadline smaller than the second smallest value, 17.1, in the template. Since $\{9, 10, 2, 8\} \not\preceq \{9, 13, 10, 8\}$, there is no schedule for this choice of anchored gaps.

We next discuss how the existence of best r-sets relates to the existence of best d-sets. Given a set J of jobs that constitute a problem P, the *mirror-image* problem P^M is defined as follows. For each job i in J with release time r_i and deadline d_i, there is a job i in J^M with release time $r_i^M = r_{\min} + (d_{\max} - d_i)$ and deadline $d_i^M = d_{\max} - (r_i - r_{\min})$. It would be convenient if i were in the best d-set in P^M if and only if i were in the best r-set in P. Unfortunately, this is not the case, since in both problems P and P^M, the relation $\preceq$ is applied to sets of deadlines. We show the existence of the best d-set below in Theorem 3.3, and also show how to generate it using a modified mirror-image problem.

THEOREM 3.3 (best d-set). *Let $[a, b)$ be an interval. Let J be an $[a, b)$-feasible set of jobs that is simple. Then there is a best d-set for $[a, b)$ with respect to J.*

Proof. Let $[a, b)$ be an interval, J an $[a, b)$-feasible set of jobs, and let P be a *d-problem* consisting of finding the best d-set for the interval $[a, b)$ with respect to J, if such a set exists. We generate a new d-problem $\widehat{P}$ with job set $\widehat{J}$ in the following way. For each job $j \in J$ with release time r_j and deadline d_j there is a job j in $\widehat{J}$ with deadline $\widehat{d_j} = d_j$ and release time $\widehat{r_j} = r_j$ if $r_j \geq a$, and $\widehat{r_j} = a - (\widehat{d_j} - a)$ otherwise. It follows that the release times $\widehat{r_j}$ in $\widehat{P}$ are all distinct, as are the deadlines $\widehat{d_j}$. The d-problem $\widehat{P}$ is to find the best d-set for the interval $[a, b)$ with respect to $\widehat{J}$. A set is the best d-set for P if and only if it is the best d-set for $\widehat{P}$, since the only differences in jobs in J and $\widehat{J}$ are modified release times that are outside of the interval $[a, b)$. The release times are modified in such a way that for any two jobs j and k in J with $r_j, r_k < a$, if $d_j < d_k$, then $\widehat{r_j} > \widehat{r_k}$.

Let $\widehat{P}^M$ be an r-problem formed by taking the mirror image of d-problem $\widehat{P}$, as follows. For each job j let release time $r_j^M = \widehat{r}_{\min} + (\widehat{d}_{\max} - \widehat{d_i})$ and deadline $d_i^M = \widehat{d}_{\max} - (\widehat{r_i} - \widehat{r}_{\min})$. Let $a^M = \widehat{r}_{\min} + (\widehat{d}_{\max} - b)$ and $b^M = \widehat{d}_{\max} - (a - \widehat{r}_{\min})$. Since J is $[a, b)$-feasible, $\widehat{J}$ is $[a, b)$-feasible, and $\widehat{J}^M$ is $[a^M, b^M)$-feasible. Thus there is an r-set for $[a^M, b^M)$ with respect to $\widehat{J}^M$. Since the release times $\widehat{r_j}$ in $\widehat{P}$ are distinct, and the deadlines $\widehat{d_j}$ are also distinct, it follows that the release times r_j^M in $\widehat{P}^M$ are distinct, and the deadlines d_j^M are also distinct. By Theorem 3.1, there is a best r-set for $[a^M, b^M)$ with respect to $\widehat{J}^M$. This best r-set is equivalent to the best d-set for $[a, b)$ with respect to $\widehat{J}$, which is equivalent to the best d-set for $[a, b)$ with respect to J. □

The proof of Theorem 3.3 identifies a method for computing d-sets. When an interval $[a, b)$ does not contain a prime interval, computing the best d-set for it is easy. As in the previous proof, modify the release times of the jobs, transform the problem into a mirror-image problem, and compute an r-set using the discrete earliest deadline rule. This best r-set is equivalent to the best d-set in the original problem.

When an interval $[a, b)$ does contain a prime interval, we can construct its best d-set by combining the best d-sets of two subproblems. We need a lemma similar to Theorem 3.2, which applied to r-sets. Let J be a set of jobs whose maximum deadline is $d_{\max}$, where $d_{\max} > b$. We wish to consider the case when jobs in $J_d[b, d_{\max})$ would be forced to complete by time b. Let $\tilde{b} = b + \lfloor d_{\max} - b \rfloor + 1$. Let the *b-extension of* J be the set J^b of jobs $\{j_i \mid i = 1, \ldots, \tilde{b} - b\}$ with $d_{j_i'} = b + i$ and $r_{j_i'} = d_{j_i'} - 1$. When $J \cup J^b$ is $[a, \tilde{b})$-feasible, any schedule must have the interval $[b, \tilde{b})$ scheduled with jobs only in J^b, and all jobs in J must be scheduled to complete by b. The set J^b is an artifice that allows us to conveniently discuss how a best choice of a set scheduled for $[a, b)$ would differ from a best d-set for $[a, b)$.

LEMMA 3.3. *Let $[a, b)$ be an interval, and J a set of jobs such that $J \cup J^b$ is $[a, \tilde{b})$-feasible and simple. Let A_1 be the best d-set for $[a, b)$ with respect to J, and let A_2 be the best d-set for $[a, \tilde{b})$ with respect to $J \cup J^b$. Then $A_2 - J^b \subseteq A_1 \cup J_d[b, d_{\max})$.*

Proof. The proof is similar to that of Theorem 3.3 except that it appeals to Theorem 3.2 rather than Theorem 3.1. Let $\widehat{J'}$ be the subset of J whose deadlines are greater than a, with release times and deadlines modified as follows. For each job $j \in J$ with $d_j \geq a$, there is a job j in $\widehat{J'}$ with deadline $\widehat{d_j} = \min\{d_j, b\}$ and release time $\widehat{r_j} = r_j$ if $r_j \geq a$, and $\widehat{r_j} = a - (d_j - a)$ otherwise. Note that all $\widehat{r_j}$ will be distinct. Let $\widehat{J}$ be the subset of $\widehat{J'}$ corresponding to the subset $J_d[a, b)$ of J. Let $a^M = \widehat{r}_{\min} + (\widehat{d}_{\max} - b)$ and $b^M = \widehat{d}_{\max} - (a - \widehat{r}_{\min})$. Let $\widehat{J'}^M$ be the mirror-image set of jobs corresponding to $\widehat{J'}$, and $\widehat{J}^M$ be the mirror-image subset of jobs corresponding to $\widehat{J}$. Since $J \cup J^b$ is $[a, \tilde{b})$-feasible, $\widehat{J'}$ is $[a, b)$-feasible, and $\widehat{J'}^M$ is $[a^M, b^M)$-feasible. Thus $\widehat{J}^M$ is $[a^M, b^M)$-feasible, and there is an r-set for $[a^M, b^M)$ with respect to $\widehat{J}^M$. Note that all $\widehat{d}_j^M$ will be distinct, since all $\widehat{r_j}$ are distinct. By Corollary 3.1.1, there is a best r-set $\widehat{A}_1^M$ for $[a^M, b^M)$ with respect to $\widehat{J}^M$, and a best r-set $\widehat{A}_2^M$ for $[a^M, b^M)$ with respect to $\widehat{J'}^M$. By Theorem 3.2, $\widehat{A}_2^M \subseteq \widehat{A}_1^M \cup (\widehat{J'}^M - \widehat{J}^M)$. Best r-set $\widehat{A}_1^M$ is equivalent to best d-set $\widehat{A}_1$ for $[a, b)$ with respect to $\widehat{J}$, which is equivalent to best d-set A_1 for $[a, b)$ with respect to J. Best r-set $\widehat{A}_2^M$ is equivalent to a subset $\widehat{A}_2 \subseteq \widehat{J'}$. Since the best d-set A_2 for $[a, \tilde{b})$ with respect to $J \cup J^b$ must contain all of J^b, and any schedule for A_2 must schedule every job in $A_2 - J^b$ to complete by b, $\widehat{A}_2$ is equivalent to $A_2 - J^b$. The theorem then follows. □

The computation of best d-sets for intervals that contain prime intervals is similar to the computation of r-sets, and will be discussed in more detail in the next section.

4. The NC algorithm. In this section we describe a parallel divide-and-conquer algorithm on a CREW PRAM for determining if there is a schedule, and if so, generating it. The algorithm consists of four steps, plus a preprocessing step. The preprocessing step replaces the original set of jobs with an equivalent set of jobs, in which all release times are distinct, all deadlines are distinct, and each deadline is a breakpoint. The first step uses the characterization of §2 to form a cover and to label the jobs as either cover jobs or gap jobs. Then the jobs are partitioned, with each job assigned to either a cover interval or a gap based on release times. A second partition is also generated, based on deadlines. The second step imposes a balanced binary tree structure on the problem, with the leaves representing gaps in order from earliest to latest, and with each nonleaf node representing a multiple gap containing the gaps represented by its leaf descendants. The characterizations of §3 are used to compute best r-sets and best d-sets for anchored gaps and anchored multiple gaps corresponding to the tree nodes. If there is a best r-set that includes all jobs, then there is a schedule; otherwise the algorithm halts. If a schedule exists, then the third step obtains a schedule of the gap jobs by starting with the largest anchored multiple gap and its best r-set, and repeatedly splitting anchored multiple gaps and their corresponding set of jobs into two constituent anchored multiple gaps and a cover interval, until only anchored gaps and their corresponding sets remain. The jobs within each of these sets can easily be scheduled within their assigned anchored gap. Given the endpoints of the anchored gaps, the fourth step schedules the cover jobs in the cover intervals, using Lemma 2.1. We discuss each step carefully and analyze its time and processor requirements.

The preprocessing step first makes every deadline a breakpoint. For each job j, the fractional parts $u_j = r_j - \lfloor r_j \rfloor$ and $v_j = d_j - \lfloor d_j \rfloor$ of its release time and deadline are determined. The multiset of values u_j is then sorted. For each j, a binary search is performed in the sorted list to find the largest u_i no larger than v_j. (If there is no such u_i, then u_i is taken to be -1 plus the largest value in the list.) Then d_j is reset to be $d_j - v_j + u_i$.

Next the preprocessing step increases certain release times so that all release times are distinct, and decreases certain deadlines so that all deadlines are distinct. As stated in §3, we want to perform the following operations repeatedly, while they apply. If $r_j = r_k$ and $d_j \leq d_k$ for jobs j and k, then reset r_k to be $r_j + 1$. If $r_j < r_k$ and $d_j = d_k$, then reset d_j to be $d_k - 1$. We first describe how to handle all instances of the release time modifications in parallel, and then all instances of the deadline modifications.

First we describe how to make all release times distinct. For each release time r_j, its fractional part $u_j = r_j - \lfloor r_j \rfloor$ is extracted. The multiset of the values u_j are then sorted. Then the jobs j are partitioned into sets R_u such that $u_j = u$. For each set R_u, the following is done. The parallel version of the discrete earliest deadline first rule [AGK, R] is applied to the set. If no schedule is possible, then our algorithm halts with failure. Otherwise reset the release time of job j to be its starting time in the schedule.

To make all deadlines distinct first convert the problem into a mirror-image problem. The release time r_j^M for job j in P^M is set to $r_{\min} + (d_{\max} - d_j)$, and the deadline d_j^M for job j in J^M is set to $d_{\max} - (r_j - r_{\min})$. Then the above algorithm for resetting release times is run, but without resetting the release times. Instead, the deadline of job j is reset to $d_j - (s_j - (r_{\min} + d_{\max} - d_j)) = d_{\max} + r_{\min} - s_j$, where s_j is the starting time of job j in the corresponding schedule for the mirror-image problem.

LEMMA 4.1. *Given a set J_0 of n unit-time jobs with arbitrary release times and deadlines, an equivalent simple set J with all deadlines being breakpoints can be found in $O(\log n)$ time using $O(n)$ processors.*

Proof. The correctness of the above procedure is established as follows. By Theorem 2.1, deadlines need only be breakpoints. Next consider generating simple set J. When applying the first of the above operations, the only jobs that can have equal release times at some point are those that have an equal fractional part. The discrete earliest deadline rule schedules correctly for any set of jobs, all of whose release times differ from some value by an integral amount. The discrete earliest deadline rule always schedules a job at the earliest possible release time, subject to no other job being available and having an earlier deadline. Thus the starting time of the job corresponds to the release time generated by the repeated application of the above operation. The application of the discrete earliest deadline rule to the mirror-image problem gives a schedule in which every job starts as late as possible. Correctness then follows.

We analyze the resource bounds as follows. Sorting will use $O(\log n)$ time on $O(n)$ processors, and performing n binary searches in parallel, as well as the parallel version of the discrete earliest deadline rule [AGK], [R], will use the same resources. □

The first step identifies subproblems that can be solved independently. The subproblems are formed by finding a cover and its associated gaps, and partitioning the jobs into sets that are associated with either a cover interval or a gap. The cover is found by forming the set of all possible constrained intervals and then deleting those that are neither prime nor compatible. First the constrained intervals are identified. For each pair consisting of a release time r_i and a deadline d_j, where $r_i < d_j$, let $n_{i,j}$ be the number of jobs contained in this interval. If $d_j - r_i < n_{i,j}$, then the algorithm halts, as no schedule exists. If $d_j - r_i - n_{i,j} < 1$, then $[r_i, d_j)$ is a constrained interval. Second, for each release time r_i, if there is more than one constrained interval starting at r_i, then all such intervals except for the one with the smallest deadline are deleted. Similarly, for each deadline d_j, if there is more than one constrained interval ending at d_j, then all such intervals except for the one with the largest release time are deleted. At most n constrained intervals will remain.

Third, the prime intervals are identified. Each constrained interval is compared with every other constrained interval and deleted if it contains such an interval. The fourth step is to form a cover. The prime intervals $[a_i, b_i)$ are sorted on the values a_i. Since no interval is contained in another, they are also sorted by b_i. For each $[a_i, b_i)$ binary search is used to determine the prime interval $[a_j, b_j)$ with $i < j$ such that $[a_i, b_i)$ and $[a_j, b_j)$ are compatible and for any k, $i < k < j$, $[a_i, b_i)$ and $[a_k, b_k)$ are not compatible. Using recursive doubling, a maximal set of prime intervals that are compatible is identified, and those prime intervals that are not compatible with one of the selected prime intervals are deleted. The remaining prime intervals constitute a cover.

Having identified a cover, the gaps are then identified. The set of jobs are then partitioned in the two partitions as follows. Any job contained in a cover interval is a cover job, and is assigned to the cover interval in both partitions. The remaining jobs are gap jobs, and are assigned as follows. For the partition based on release times, if the release time of a gap job falls within a gap, then the job is assigned to that gap. Otherwise, the gap job is called *anomalous*, and it is assigned to the cover interval containing its release time. For the partition based on deadlines, if the deadline of a gap job falls within a gap, then the job is assigned to that gap; otherwise, the gap job is assigned to the cover interval containing its deadline.

LEMMA 4.2. *Given a set of n unit-time jobs with arbitrary release times and deadlines, a cover can be computed in $O(\log n)$ time using $O(n^2/\log n)$ processors.*

Proof. The correctness of the above procedure follows from the definition of a cover. We next analyze the time complexity. Assuming that jobs are indexed by nondecreasing deadlines, the $n_{i,j}$ are computed as follows. Let $v_{i,j}$ indicate whether job j lies within $[r_i, d_j)$, so $v_{i,j} = 1$ if $r_j \geq r_i$ and $v_{i,j} = 0$ if $r_j < r_i$. The $n_{i,j}$ are computed by performing a prefix sum over the $v_{i,j}$'s, $n_{i,j} = v_{i,1} + v_{i,2} + \ldots + v_{i,j}$. Using Brent's Theorem [B], this uses $O(n^2/\log n)$ processors. Reducing the number of constrained intervals under consideration to at most n also uses $O(n^2/\log n)$ processors. Assigning one processor to $\log n$ pairs of constrained intervals, identifying prime intervals and finding a compatible set uses $O(n^2/\log n)$ processors. Each of the above activities uses $O(\log n)$ time. □

We next discuss the second step in our algorithm. It first imposes a balanced binary tree structure on the problem, with the leaves representing gaps in order from earliest to latest, and with each nonleaf node representing a multiple gap containing the gaps represented by its leaf descendants. It then computes best r-sets and best d-sets, if they exist, for anchored gaps and anchored multiple gaps corresponding to the tree nodes, using a bottom-up sweep through the tree. The final result at the root of the tree will be the best r-set and the best d-set for $[r_{\min}, d_{\max})$. The best r-sets are in sorted order by deadlines, not in scheduled order, so that they can be merged with other best r-sets quickly. Best d-sets are also in sorted order by deadlines.

We first discuss computing best r-sets and best d-sets for anchored gaps. Let $[a, b)$ be a gap, preceded by a cover interval of looseness x and followed by a cover interval of looseness x'. Associated with gap $[a, b)$ are the anchored gaps $[a_i, b_h)$, where $a_i \in [a, a + x)$, and

$b_h \in (b - x', b]$, and a_i and b_h are breakpoints. Assume that the a_i, and also the b_h, are indexed in increasing order. Thus there are at most n^2 anchored gaps associated with each gap. Consider one anchored gap $[a_i, b_h)$ associated with gap $[a, b)$. Recall that $J_r[a, b)$ is the set of gap jobs assigned to gap $[a, b)$ in the release time partition. We shall understand $J_r[a_i, b_h)$ to be $J_r[a, b)$. (For any job assigned to $[a, b)$ whose release time is less than a_i, we are implicitly assuming that its release time is modified to be a_i for anchored gap $[a_i, b_h)$, for the purposes of computing best r-sets. We make a similar assumption for any anchored multiple gap that starts at a_i.) A similar understanding applies for $J_d[a_i, b_h)$ and the deadline partition. For each anchored gap $[a_i, b_h)$ and set of jobs $J_r[a_i, b_h)$, the best r-set for $[a_i, b_h)$, denoted $J_r^*[a_i, b_h)$, is computed. Those jobs from $J_r[a_i, b_h)$ that are not chosen for $J_r^*[a_i, b_h)$ are placed into $J_r^-[a_i, b_h)$, the set of remaining jobs not chosen yet. Similarly, considering the jobs $J_d[a_i, b_h)$, the best d-set for $[a_i, b_h)$, denoted $J_d^*[a_i, b_h)$, is computed and the remaining jobs from $J_d[a_i, b_h)$ not chosen for $J_d^*[a_i, b_h)$ are placed into $J_d^-[a_i, b_h)$.

There is no need to compute the best r-set for each of the at most n^2 anchored gaps, since there are at most $2n$ distinct best r-sets for these anchored gaps. For a given b_h and all possible a_i, there are at most two distinct best r-sets for all of the corresponding anchored gaps $[a_i, b_h)$. All these intervals consider the same set of jobs, but the larger intervals might be able to schedule one more job than the smaller intervals. This follows since $\lfloor b_h - a_1 \rfloor \leq \lfloor b_h - a_k \rfloor + 1$, where a_k is the largest a_i. If there are two best r-sets, then there is some a_l such that all $[a_i, b_h)$, $i \leq l$, have the same best r-set, and all $[a_i, b_h)$, $i > l$, have the other best r-set. The two best r-sets can be found by computing best r-sets for $[a_1, b_h)$ and $[a_k, b_h)$ using the discrete earliest deadline rule. Computing at most $2n$ best r-sets instead of n^2 best r-sets reduces the number of processors needed for this activity by a factor of n.

We show how to compute best r-sets for anchored gaps contained in gap $[a, b)$. First compute $J_r^*[a_1, b_h)$ for all valid indices h. Apply the parallel version of the discrete earliest deadline rule [AGK], [R] for the jobs in $J_r[a_1, b_h)$ and the interval $[a_1, b_h)$. Set $J_r^-[a_1, b_h)$ to $J_r[a_1, b_h) - J_r^*[a_1, b_h)$. In the same manner, compute $J_r^*[a_k, b_h)$ and $J_r^-[a_k, b_h)$ for all valid indices h, where k is the largest index for the a_i. The set $J_r^*[a_i, b_h)$ is set to $J_r^*[a_1, b_h)$ if $\lfloor b_h - a_i \rfloor = \lfloor b_h - a_1 \rfloor$; otherwise it is set to $J_r^*[a_k, b_h)$. In the first case, $J_r^-[a_i, b_h) = J_r^-[a_1, b_h)$ and in the second case $J_r^-[a_i, b_h) = J_r^-[a_k, b_h)$. These additional best r-sets and remaining sets do not need to be computed as they are just duplicates of other sets.

In a similar manner, the best d-sets for anchored gaps contained in gap $[a, b)$ are computed. For a given a_i, there are at most two best d-sets. The release times of each job $j \in J_d[a_i, b_h)$ with $r_j < a_i$ is reset to $a_i - (d_j - a_i)$, the mirror image of this problem is formed, and then solved by the discrete earliest deadline first rule. The maximum number of jobs that can be scheduled in $[a_i, b_h)$ is stored as $C[a_i, b_h)$. For anchored gap $[a_i, b_h)$, this value is $\lfloor b_h - a_i \rfloor$.

LEMMA 4.3. *For all anchored gaps, determining best r-sets and best d-sets whenever they exist uses $O(\log n)$ time and $O(n^2)$ processors.*

Proof. Correctness of the above procedure follows from Lemma 3.1, Theorem 3.3, and the above discussion. To analyze the time and processor complexity, let the lth gap have n_l jobs associated with it. A best r-set or best d-set is computed for at most $2n$ anchored gaps. The parallel version of the discrete earliest deadline rule [AGK], [R] uses $O(\log n)$ time and $O(n)$ processors to schedule n jobs. Thus computing one best r-set or one best d-set in parallel takes $O(\log n_l)$ time and $O(n_l)$ processors, so the total number of processors needed to compute the $2n$ best r-sets for the lth gap is $O(n * n_l)$. Since $n = n_1 + n_2 + \ldots + n_g$, the total number of processors for all gaps is $O(n^2)$. □

We next discuss computing best r-sets and best d-sets for anchored multiple gaps. Let $[a, b')$ be a multiple gap composed of the two consecutive gaps and/or multiple gaps $[a, b)$ and $[a', b')$. In general multiple gaps overlap their two surrounding cover intervals, so that the cover interval between $[a, b)$ and $[a', b')$ is not $[b, a')$. For convenience we shall abuse our notation slightly and refer to this cover interval as $[b, a')$. Assume that best r-sets and best d-sets have already been computed for all anchored multiple gaps associated with $[a, b)$ and $[a', b')$. Let $[a_i, b_h)$ be an anchored multiple gap for $[a, b)$, and let $[a'_g, b'_f)$ be an anchored multiple gap for $[a', b')$, where $a'_g - b_h$ equals the number of cover jobs contained in the cover interval $[b, a')$, and such that $J_r^*[a_i, b_h)$ and $J_r^*[a'_g, b'_f)$ exist. Among r-sets of $[a_i, b'_f)$ that have a schedule in which a cover job starts at b_h, if one exists, let $J_r^{*h}[a_i, b'_f)$ be the best such r-set for $[a_i, b'_f)$. Define $J_d^{*h}[a_i, b'_f)$ similarly. Let $J_r^{-h}[a_i, b'_f) = J_r[a_i, b'_f) - J_r^{*h}[a_i, b'_f)$, and similarly for $J_d^{-h}[a_i, b'_f)$. For every pair $[a_i, b_h)$ and $[a'_g, b'_f)$ such that $a'_g - b_h$ equals the number of cover jobs contained in $[b, a')$, sets $J_r^{*h}[a_i, b'_f)$ and $J_d^{*h}[a_i, b'_f)$ are computed if they exist. Then, for each anchored multiple gap $[a_i, b'_f)$ of multiple gap $[a, b')$, the various values of b_h are examined, and from among the corresponding $J_r^{*h}[a_i, b'_f)$, if there are any, the best r-set is identified as $J_r^*[a_i, b'_f)$. Then $J_r^-[a_i, b'_f)$ is chosen to be the corresponding $J_r^{-h}[a_i, b'_f)$. Similarly, from among the corresponding $J_d^{*h}[a_i, b'_f)$, if there are any, the best d-set is identified as $J_d^*[a_i, b'_f)$. The value $C[a_i, b'_f)$ is set to $C[a_i, b_h) + (a'_g - b_h) + C[a'_g, b'_f)$.

We next discuss how to compute $J_r^{*h}[a_i, b'_f)$ if it exists. Note that $J_r^*[a_i, b_h) \subseteq J_r^{*h}[a_i, b'_f)$, since the jobs in $J_r^*[a_i, b_h)$ can be scheduled in the $[a_i, b_h)$ portion of $[a_i, b'_f)$, and none of the jobs from $J_r^*[a'_g, b'_f)$ or $J_r^-[a'_g, b'_f)$ can be considered in $[a_i, b_h)$. Since the set of cover jobs of a cover interval is unaffected by the exact positioning of the gaps on either side of it, we let $J^*[b, a')$ be the set of cover jobs contained in $[b, a')$ and let $J_r^-[b, a')$ be the set of anomalous gap jobs whose release times lie in $[b, a')$. Clearly, $J^*[b, a') \subseteq J_r^{*h}[a_i, b'_f)$. We now focus on computing the jobs for the $[a'_g, b'_f)$ portion of $J_r^{*h}[a_i, b'_f)$. Let $J' = J_r^*[a'_g, b'_f) \cup J_r^-[a_i, b_h) \cup J_r^-[b, a')$. Let J'' be the $\min\{|J'|, C[a'_g, b'_f)\}$ jobs with smallest deadlines from set J'. If $d(J_d^*[a'_g, b'_f)) \preceq d(J')$ and there are no more than $|J_d^*[a'_g, b'_f)|$ jobs in J' with deadlines at most b'_f, then the jobs in J'' can be scheduled within $[a'_g, b'_f)$. In this case, $J_r^{*h}[a_i, b'_f) = J_r^*[a_i, b_h) \cup J^*[b, a') \cup J''$, and $J_r^{-h}[a_i, b'_f) = J_r^-[a'_g, b'_f) \cup (J' - J'')$. Otherwise there is no schedule using anchored multiple gap $[a_i, b'_f)$ and starting a cover job at b_h.

We next discuss how to compute $J_d^{*h}[a_i, b'_f)$ if it exists. Clearly, $J_d^*[a'_g, b'_f) \subseteq J_d^{*h}[a_i, b'_f)$, and $J^*[b, a') \subseteq J_d^{*h}[a_i, b'_f)$. We now focus on computing the jobs for the $[a_i, b_h)$ portion of $J_d^{*h}[a_i, b'_f)$. Let $J' = J_d^*[a_i, b_h) \cup J_d^-[a'_g, b'_f) \cup J_d^-[b, a')$. A set $\widehat{J}$ with the following modified release times is generated as follows. For each job $j \in J'$, let $\hat{r}_j$ be the modified release time and let r_j be the original release time. If r_j is less than a_i then $\hat{r}_j$ is set to $a_i - (d_j - a_i)$, otherwise $\hat{r}_j$ is set to r_j. Then $\widehat{J}$ is sorted by modified release times. The $\min\{|J'|, C[a_i, b_h)\}$ jobs in $\widehat{J}$ with largest modified release times are identified, with J'' being the set of corresponding jobs in J'. If $J_r^{*h}[a_i, b'_f)$ exists, then $J_d^{*h}[a_i, b'_f) = J'' \cup J_d^*[a'_g, b'_f) \cup J^*[b, a')$, and $J_d^{-h}[a_i, b'_f) = J_d^-[a_i, b_h) \cup (J' - J'')$. Otherwise, there is no set $J_d^{*h}[a_i, b'_f)$.

This step is complete when it has been determined if there is an r-set for the root of the tree. If there is an r-set for $[r_{\min}, d_{\max})$, then a schedule of the jobs exists.

LEMMA 4.4. *For all anchored multiple gaps, computing best r-sets and best d-sets whenever they exist uses $O((\log n)^2)$ time and $O(n^4/\log n)$ processors.*

Proof. As the crucial step in computing $J_r^{*h}[a_i, b'_f)$, consider the computation of J''. Since none of the jobs in J' can be scheduled in $[a_i, b_h)$, those jobs with release times less than a'_g can have their release times reset to a'_g. By Theorem 3.2, if J' is $[a'_g, b'_f)$-feasible, then there is a best r-set for $[a'_g, b'_f)$ that will contain no jobs from $J_r^-[a'_g, b'_f)$. It follows that the computation of $J_r^{*h}[a_i, b'_f)$ is correct. A similar argument using Lemma 3.3 establishes the correctness of the computation of $J_d^{*h}[a_i, b'_f)$.

We next consider time and processor complexity. Let $[a, b)$ and $[a', b')$ be the multiple gaps associated with two sibling nodes, and let there be n_l gap jobs whose release time lies within $[a, b')$. There are at most n^2 anchored multiple gaps $[a_i, b_h)$ for $[a, b)$. Each anchored multiple gap $[a_i, b_h)$ must be matched against an anchored multiple gap $[a'_g, b'_f)$ for $[a', b')$, where $a'_g - b_h$ equals the number of cover jobs that lie within the cover interval that lies between these anchored multiple gaps. Thus for any anchored multiple gap $[a_i, b_h)$, there are at most n anchored multiple gaps $[a'_g, b'_f)$ with which it must be checked. Thus at most n^3 pairs of anchored multiple gaps $[a_i, b_h)$ and $[a'_g, b'_f)$ must be checked. Each pair of anchored multiple gaps $[a_i, b_h)$ and $[a'_g, b'_f)$ can be checked in $O(\log n)$ time using $n_l / \log n$ processors. Since the sum of n_l for all gaps at one level of the tree is $O(n)$, the total number of processors needed is $n^4 / \log n$. Since there are at most $\log n$ levels in the tree, the total time for this activity is $O((\log n)^2)$. □

We next discuss the third step in our algorithm. With respect to the balanced binary tree structure imposed on the previous step, this step selects one anchored multiple gap or anchored gap for each node in the tree, along with a corresponding set of jobs, using a top-down pass through the tree. We call these anchored multiple gaps and anchored gaps *selected multiple gaps* and *selected gaps*, respectively. We call the set $J^*[a_i, b_h)$ that corresponds to a selected multiple gap or selected gap $[a_i, b_h)$ the *selected set* for $[a_i, b_h)$. The final result is a partition of the gap jobs into selected sets for anchored gaps, from which a schedule of the gap jobs within selected gaps is obtained.

This step begins by noting that $[r_{\min}, d_{\max})$ is a selected multiple gap, and the best r-set for $[r_{\min}, d_{\max})$ is a selected set for $[r_{\min}, d_{\max})$. Let $[a, b')$ be a multiple gap, and $[a, b)$ and $[a', b')$ its two constituent gaps or multiple gaps, based on the structure of the binary tree. Let $[a_i, b'_f)$ be the selected multiple gap for $[a, b')$. Let $[a_i, b_h)$ and $[a'_g, b'_f)$ be anchored multiple gaps for the multiple gaps $[a, b)$ and $[a', b')$, respectively. This step considers the at most n pairs such that $a'_g - b_h$ is equal to the number of cover jobs that are contained in the cover interval that lies between $[a_i, b_h)$ and $[a'_g, b'_f)$. Each pair $[a_i, b_h)$ and $[a'_g, b'_f)$ is examined to determine if there is a partition of $J^*[a_i, b'_f)$ into two sets such that the first can be scheduled in $[a_i, b_h)$ and the second can be scheduled in $[a'_g, b'_f)$. A pair is then chosen for which there is such a partition, with the first set being $J^*[a_i, b_h)$, and the second set being $J^*[a'_g, b'_f)$.

We next discuss how to test a given pair $[a_i, b_h)$ and $[a'_g, b'_f)$ to see if there is the desired partition of $J^*[a_i, b'_f)$. First, all gap jobs in $J^*[a_i, b'_f)$ that could be scheduled in $[a_i, b_h)$ are identified. Let J' be the set of all jobs in $J^*[a_i, b'_f)$ whose release times are less than a_i. Let $J'' = (J^*[a_i, b'_f) \cap J^*_r[a_i, b_h)) \cup J'$. Let J''' be the $\min\{|J''|, C[a_i, b_h)\}$ jobs of J'' with smallest deadlines. If $d(J^*_d[a_i, b_h)) \preceq d(J''')$, then the jobs in J''' can be scheduled within $[a_i, b_h)$. Let $J'''' = J^*[a_i, b'_f) - J^*[b, a') - J'''$ be the set of jobs that must be scheduled in $[a'_g, b'_f)$. If $d(J^*_d[a'_g, b'_f)) \preceq d(J'''')$ and the number of jobs in J''' with deadline at most b'_f is at most $|J^*_d[a'_g, b'_f)|$ and $|J''''| \leq C[a'_g, b'_f)$, then the jobs in J'''' can be scheduled within $[a'_g, b'_f)$. If both tests succeed then $[a_i, b_h)$ and $[a'_g, b'_f)$ can be selected, and $J^*[a_i, b_h)$ and $J^*[a'_g, b'_f)$ would be J''' and J'''', respectively.

LEMMA 4.5. *Given best r-sets and best d-sets for all anchored gaps and anchored multiple gaps for which these sets exist, identifying selected gaps and selected multiple gaps and the corresponding selected sets uses* $O((\log n)^2)$ *time and* $O(n^2 / \log n)$ *processors.*

Proof. We first discuss correctness. Since $J^*[a_i, b'_f)$ exists, there must be some choice of b_h such that there is a partition of $J^*[a_i, b'_f) - J^*[b, a')$ into two sets that can be scheduled in $[a_i, b_h)$ and $[a'_g, b'_f)$, respectively. Suppose we are examining such a choice b_h. We verify that we make this partition correctly. It does not alter the results to assume that any job in J' has modified release time a_i. By Theorem 3.2, there is a best r-set for $[a_i, b_h)$ with respect to J (with modified release times) that contains only jobs in $J' \cup J^*_r[a_i, b_h)$. By the approach that we employ in the algorithm, a job in $J_r^*[a_i, b_h)$ will be found in $J^*[a_i, b'_f)$ unless there is a corresponding job with release time before a that has replaced it. Thus $|J''| \geq |J_r^*[a_i, b_h)|$. Thus given $J^*[a_i, b'_f)$, our algorithm makes a best choice of jobs to be scheduled in $[a_i, b_h)$.

We next analyze the time and processors used to determine $J^*[a_i, b_h)$ and $J^*[a'_g, b'_f)$. At most n pairs $[a_i, b_h)$ and $[a'_g, b'_f)$ are checked, and these can all be checked in $O(\log n)$ time using $O(n\ |J^*[a_i, b'_f)|\ / \log n)$ processors. Thus $O(n^2/ \log n)$ processors are used for one level in the tree. The time for each level is $O(\log n)$ and there are at most $\log n$ levels in the tree, so the total time is $O((\log n)^2)$. □

We next discuss scheduling the gap jobs in gaps. For each gap, there is a selected gap and a corresponding selected set. Unfortunately, the jobs in the selected set are in sorted order by deadlines, not in scheduled order. The schedule of jobs in the selected set is computed using a modified earliest deadline algorithm.

Let $[a_i, b_h)$ be the selected gap of gap $[a, b)$, with corresponding selected set $J^*[a_i, b_h)$. The location of the first hole in $[a_i, b_h)$ is determined, and $J^*[a_i, b_h)$ is partitioned into two sets of jobs, those jobs that will be scheduled before the first hole in the gap, and those jobs that will be scheduled after the first hole. If $|J^*[a_i, b_h)| = \lfloor b_h - a_i \rfloor$, then there is only one hole, otherwise there can be several holes. To compute the position of the first hole, the smallest c is found such that there is no job available to be scheduled at position $a_i + c$. Thus the starting position of the first hole is at $a_i + c$. The c jobs with release times at most $a_i + c - 1$ are scheduled, using the discrete earliest deadline algorithm [AGK], [R]. Then the remaining $\lfloor b_h - a_i \rfloor - c$ jobs in $[b_h - (\lfloor b_h - a_i \rfloor - c), b_h)$ are scheduled, again using the discrete earliest deadline algorithm.

LEMMA 4.6. *Computing the schedule of the jobs in selected sets within their selected gaps uses $O(\log n)$ time and $O(n)$ processors.*

Proof. First we discuss correctness. By definition of selected set, a schedule exists. There are c jobs with release time at most $a_i + c - 1$. The discrete earliest deadline algorithm schedules these jobs in $[a_i, a_i + c)$. The remaining jobs must be scheduled in $[a_i + c, b_h)$. Since there is a schedule of these jobs within this interval, we may reset any deadline greater than b_h to be b_h. If any constrained interval is created, it must end at b_h. Consider such a constrained interval $[a'', b_h)$ that contains no other such constrained interval. Only the jobs contained in this interval can be scheduled so as to overlap with interval $[b_h - \lfloor b_h - a'' \rfloor, b_h)$. By Lemma 2.1, these jobs can be scheduled in $[b_h - \lfloor b_h - a'' \rfloor, b_h)$. The same argument may be applied inductively to any remaining jobs and interval $[a_i + c, b_h - \lfloor b_h - a'' \rfloor)$. All such jobs scheduled start at a time that differs from b_h by an integer. When no constrained intervals remain, the remaining jobs can be scheduled by the discrete earliest deadline rule to also satisfy this requirement. But then there exists a schedule of remaining jobs in which all jobs start at a time that differs from b_h by an integer. The discrete earliest deadline rule will find such a schedule.

Next we analyze the time and processors used. The first hole in the lth gap is computed by sorting the jobs in nondecreasing release time order, and performing a prefix sum on the number of release times less than or equal to unit spaced positions in the interval. This takes $O(\log n_l)$ time and $O(n_l)$ processors, where n_l is the number of jobs to be scheduled in the selected gap. Applying the parallel earliest deadline algorithm to the two pieces of the gap also takes $O(\log n_l)$ time and $O(n_l)$ processors. Computing one schedule for each gap in parallel takes $O(\log n)$ time and $O(n)$ processors. □

The fourth step in the algorithm schedules the cover jobs. The choice of selected gaps dictates the position of the cover jobs within the cover intervals. By Lemma 2.1, there is a schedule with the jobs packed tightly together. Let $[a, b)$ and $[a', b')$ be consecutive gaps, and $[a_i, b_h)$ and $[a'_g, b'_f)$ the corresponding selected gaps. Consider the cover interval $[b, a')$.

Note that $a'_g - b_h = |J^*[b, a')|$. The discrete earliest deadline algorithm is applied to the set $J^*[b, a')$ in interval $[b_h, a'_g)$.

LEMMA 4.7. *Scheduling the cover jobs within the cover intervals takes $O(\log n)$ time and uses $O(n)$ processors.*

Proof. The parallel earliest deadline algorithm [AGK], [R] applied to the ith cover interval that contains n_l jobs takes $O(\log n_l)$ time and uses $O(n_l)$ processors. Thus computing the schedule for all cover intervals in parallel takes $O(\log n)$ time and $O(n)$ processors. □

We summarize the performance of the algorithm below.

THEOREM 4.1. *Given n unit-time jobs with arbitrary release times and deadlines, there is a CREW PRAM algorithm that determines if there is a schedule of the jobs on a single processor, and if so, it produces a schedule. The algorithm uses $O((\log n)^2)$ time and $O(n^4/(\log n))$ processors.*

Proof. By Theorem 2.1, if there is a schedule, then there is a canonical schedule. The discussion accompanying the description of the algorithm in this section, plus the correctness within Lemmas 4.2 through 4.7, establish that the algorithm computes a canonical schedule whenever one exists. The time and processor complexities follow from Lemmas 4.2 through 4.7. □

Suppose k is the number of distinct fractional parts of the release times. The algorithm uses fewer than $\Theta(n^4/(\log n))$ processors if k is $o(n)$. The number of starting positions for anchored gaps is thus reduced to k. If there are fewer distinct fractional parts of the deadlines, then breakpoints can be based on deadlines, as discussed at the end of §2. Let k be the minimum of the number of distinct fractional parts of release times and the number of distinct fractional parts of deadlines.

COROLLARY 4.1.1. Given n unit-time jobs with arbitrary release times and deadlines, there is a CREW PRAM algorithm that determines if there is a schedule of the jobs on a single processor, and if so, it produces a schedule. If k is the minimum of the number of distinct fractional parts of release times and the number of distinct fractional parts of deadlines, then the algorithm uses $O((\log n)^2)$ time and $O(k^3n/(\log n) + n^2/(\log n)^2)$ processors.

Proof. Finding a cover can be done in $O((\log n)^2)$ time using just $O(n^2/(\log n)^2)$ processors, by having each of the processors simulate $\log n$ processors in the algorithm stated earlier. Similarly, computing best r-sets and best d-sets for anchored gaps can be done in $O((\log n)^2)$ time using $O(kn/\log n)$ processors. Computing best r-sets and d-sets for anchored multiple gaps takes $O((\log n)^2)$ time and uses $O(k^3n/(\log n))$ processors. Determining selected gaps and selected multiple gaps uses $O(kn/\log n)$ processors and $O((\log n)^2)$ time. Scheduling the gap jobs and the cover jobs uses $O(\log n)$ time and $O(n)$ processors. □

REFERENCES

[AGK] M. J. ATALLAH, M. T. GOODRICH, AND S. R. KOSARAJU, *On the parallel complexity of evaluating some sequences for set manipulation operations*, J. Assoc. Comput. Mach., to appear.

[B] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[C] J. CARLIER, *Probleme a une machine dans le cas ou les taches ont des durees egales*, Tech. Rep., Institut de Programmation, Universite Paris VI, 1979.

[DS] E. DEKEL AND S. SAHNI, *Binary trees and parallel scheduling algorithms*, IEEE Trans. Comput., 32 (1983), pp. 307–315.

[DUW] D. DOLEV, E. UPFAL, AND M. K. WARMUTH, *The parallel complexity of scheduling with precedence constraints*, J. Parallel Distrib. Comput., 3 (1986), pp. 553–576.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, SIAM J. Comput., 6 (1977), pp. 416–426.

[GJST] M. R. Garey, D. S. Johnson, B. B. Simons, and R. E. Tarjan, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput., 10 (1981), pp. 256–269.

[HM1] D. Helmbold and E. Mayr, *Fast scheduling algorithms on parallel computers*, in Advances in Computing Research, F. Preparata, ed., JAI Press, Inc., Greenwich, CT, 1987.

[HM2] ———, *Two processor scheduling is in NC*, SIAM J. Comput., 16 (1987), pp. 747–759.

[H] W. A. Horn, *Some simple scheduling algorithms*, Naval Res. Logist. Quart., 21 (1974), pp. 177–185.

[KR] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Elsevier Science Publishers, New York, 1990, pp. 869–942.

[R] S. H. Rodger, *An optimal parallel algorithm for preemptive job scheduling that minimizes the maximum lateness*, Proc. 26th Allerton Conference on Communication, Control, and Computing, 1988, pp. 293–302.

[S1] B. B. Simons, *A fast algorithm for single processor scheduling*, in Proc. 19th IEEE Symp. on Foundations of Computer Science, 1978, pp. 246–252.

[S2] ———, *Multiprocessor scheduling of unit length jobs with arbitrary release times and deadlines*, SIAM J. Comput., 12 (1983), pp. 294–299.

[SW] B. B. Simons and M. K. Warmuth, *A fast algorithm for multiprocessor scheduling of unit-length jobs*, SIAM J. Comput., 18 (1989), pp. 690–710.

SIAM J. COMPUT.
Vol. 23, No. 1, pp. 212–225, February 1994

BROADCASTING AND GOSSIPING IN DE BRUIJN NETWORKS*

JEAN-CLAUDE BERMOND† AND PIERRE FRAIGNIAUD‡

Abstract. Communication schemes based on *store and forward* routing, in which a processor can communicate simultaneously with all its neighbors (in parallel) are considered. Moreover, the authors assume that sending a message of length L from a node to a neighbor takes time $\beta + L\tau$. The authors give efficient *broadcasting* and *gossiping* protocols for the de Bruijn networks. To do this, *arc-disjoint spanning trees* of small depth rooted at a given vertex in de Bruijn digraphs are constructed.

Key words. broadcasting, gossiping, interconnection networks, de Bruijn graphs, disjoint spanning trees

AMS subject classifications. 68E10, 94, 05C05, 05C35

1. Introduction. In the design and use of parallel computers, different elements are important. Among them are the topology of the interconnection network and the communication scheme. In this paper, we focus on communication problems. Important cases appearing in parallel algorithms are

Broadcasting: send a message from a given vertex to all the vertices, also called *OTA* (one-to-all).

Gossiping: send messages from all the vertices to all the other vertices, also called *ATA* (all-to-all) or total exchange.

An important literature on graph theory concerning this problem assumes that the communication cost is a constant and that only one port can be used by a processor at a given time (see the survey [15]). Here we consider the *store and forward* model, in which a vertex can simultaneously send and receive (eventually) *different* messages to and from all its neighbors. Indeed, this communication possibility corresponds to existing parallel computers (hypercubes and transputer-based machines). Moreover, the neighbor-to-neighbor communication time depends on a latency, or start-up time β, and on a data transfer time per element, or propagation time τ ($\frac{1}{\tau}$ is the bandwidth of a link). Sending a message of length L to a neighbor takes time $T = \beta + L\tau$ [20], [28], [31].

The hypothesis stating that processors are able to communicate simultaneously through all their ports is well known [20], [28], [31]. Many parallel computers satisfy this hypothesis. However, note that if the number of communication ports of a vertex is large, the start-up time (and in a less significant way, the propagation time) may grow with the number of ports used simultaneously [11]. We do not consider this problem in this paper. The reader is referred to [12] for a survey of broadcasting algorithms under several hypotheses. Note also that there exist other models of routing, such as *circuit-switched* or *wormhole* routing [8], [21]; however, in the case of intensive communications such as broadcasting or gossiping, they do not seem to offer any significative advantages over the *store and forward* model [30].

Clearly, the broadcasting and gossiping protocols depend on the topology of the interconnection network. The choice of this topology is critical in the design of parallel computers. Different goals can be pursued to increase the performance and minimize the cost. These goals can be expressed in terms of the graph (or digraph) that represents the interconnection network. The vertices of the graph correspond to processors, and the edges (or arcs) correspond to communication links between processors.

*Received by the editors April 15, 1991; accepted for publication (in revised form) October 12, 1992.

†I3S, Centre National de la Recherche Scientifique, Bât.4, Avenue Albert Einstein, Sophia Antipolis, 06560 Valbonne, France. This author's research was supported by the research program C3.

‡LIP-IMAG, Centre National de la Recherche Scientifique, Ecole Normale Supérieure de Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France. This author's research was supported by the research programs C3 and ANM, and by the Direction des Recherches et Etudes Techniques.

An important constraint claimed by many authors [7], [8], [17] is that very large-scale integration (VLSI) computing systems are wire limited, which corresponds to a fixed small degree in the associated graph. Furthermore, in the model of store and forward routing, the total transmission time depends on the diameter, which should be as small as possible. Thus, it is important to have graphs with a small diameter and a fixed degree. This is not the case for the hypercube, since the degree and diameter increase as the logarithm of the number of processors, or for the multidimensional meshes or torus, since they have diameters that are too large.

Two well-known topologies are the Kautz and de Bruijn networks, which have many interesting properties [6], [29]. In particular, these networks interconnect considerably more processors than the usual topologies, and they have small diameters and small and fixed degrees. Therefore the aim of this paper is to find efficient broadcasting and gossiping protocols in the de Bruijn digraph. Classical broadcasting in de Bruijn digraphs using the assumption of a constant communication time and the possibility of sending a message to at most one neighbor at any time has been considered in [5], [16].

This paper is organized as follows. In §2 we recall some definitions about directed graphs. In §3 we compute lower bounds and explain possible communication protocols for broadcasting in a given network. We show that an efficient way uses the construction of arc-disjoint spanning trees. Section 4 describes the de Bruijn graphs and digraphs and introduces some notation. Section 5 proposes arc-disjoint spanning trees of the de Bruijn digraphs, which can be used to perform asymptotically optimal broadcasting. In §6 a fast, greedy, gossiping algorithm is given. Section 7 explains how to translate the proposed algorithms from the de Bruijn digraphs to the undirected de Bruijn graphs. Finally, §8 concludes the paper.

2. Notation. A network of n processors is usually modeled by a graph or a digraph $G = (V, E)$ of order n. Three kinds of communication links are usually used.

1. Monodirectional links: messages can only be transferred in one direction. The network is modeled by a digraph.

2. Bidirectional *half duplex* links: such links can be used at a given time in at most one direction. The network is modeled by a graph.

3. Bidirectional*full duplex* links: each link can be simultaneously used in both directions. The network is modeled by a *symmetric* digraph.

In this paper we will consider digraphs only (symmetric or not), either corresponding to networks with monodirectional links, or networks with bidirectional *full duplex* links. In a digraph there is an arc from a node u to a node v only if u is able to send a message directly to v.

Let $d(u, v)$ be the distance between the vertices u and v, that is, the length of a shortest path from u to v. We will use $ecc(u)$ to denote the eccentricity of the vertex u, that is, $\max_{v \in V} d(u, v)$, and use D for the diameter of the digraph, that is, $\max_{u \in V} ecc(u)$. We will use "shortest-paths spanning tree rooted at r" to denote breadth first search tree rooted at r. In such a tree, the path from r to any other vertex v is a shortest path. Note that the depth of such a tree is $ecc(r)$.

Let $d^+(u)$ (resp. $d^-(u)$) denote the out-degree (or in-degree) of the vertex u, that is, the number of outgoing (or incoming) arcs from (or to) u. If the digraph is regular then all the vertices have the same out- and in-degree d. Otherwise $d^+_{\max}$ denotes the maximum out-degree over all the vertices (similar definitions are derived for $d^-_{\max}$, $d^+_{\min}$, and $d^-_{\min}$). Let $m^+(S, V - S)$ be the number of arcs going from S to $V - S$ and let $c_G(r) = \min_{S \neq V | r \in S} m^+(S, V - S)$. $c_G(r)$ can be regarded as the minimum number of arcs that must be deleted to make at least one vertex not reachable from r. Another interpretation of $c_G(r)$ is that there exist $c_G(r)$ arc-disjoint paths from r to any vertex of G. Moreover, $c_G(r)$ is the maximum number of such paths that are arc disjoint (Menger's theorem). Let λ be the arc connectivity of G, that is $\lambda = \min_{r \in V} c_G(r)$. Note that $d^+(r) \geq c_G(r) \geq \lambda$.

Finally, we will use $b_G(r)$ to denote the minimum time of any broadcast initiated by a vertex r of G, and use g_G for the minimum time of any gossiping on G.

3. Broadcasting in networks.

3.1. Lower bounds. Recall that the communication time to send a message of length L from a node to a neighbor is of the form $\beta + L\tau$. Following [20], [31], we can obtain two different lower bounds by considering the total start-up time or the total data transfer time. First the broadcasting time $b_G(r)$ is at least $ecc(r)\beta$. Consider now a subset S of V containing r such that $m^+(S, V - S) = c_G(r)$, and let u be a vertex of $V - S$. The total bandwidth of the arcs between S and $V - S$ is $c_G(r)/\tau$ and therefore the minimum time to send the message from r to u is at least $(L/c_G(r))\tau$.

PROPOSITION 3.1. *In a digraph G, the time to broadcast a message of length L from a node r is at least* $\max(ecc(r)\beta, (L/c_G(r))\tau)$.

For example, in a D-cube used in full duplex mode, $c_G(r) = d^+(r) = d^-(r) = D$, and $ecc(r) = D$. We obtain $b_{D-\text{cube}}(r) \geq \max(D\beta, \frac{L}{D}\tau)$, which is the bound proposed by Johnsson and Ho in [20].

3.2. Communication protocols. There exist different ways to perform broadcasting from an originator r. The efficiency of these protocols depends on the ratio $\frac{\beta}{L\tau}$. The first protocol simply uses a shortest-paths spanning tree. The broadcasting time is $ecc(r)(\beta + L\tau)$ because there exists at least one vertex at distance $ecc(r)$ from r. If $L\tau \ll \beta$, this time is of the order $\beta ecc(r)$ approaching the lower bound and so we cannot do better.

In contrast, if $L\tau \gg \beta$, this time is of the order $L\tau ecc(r)$, which is far from the lower bound $L\tau/c_G(r)$. In case of long messages, we can improve the total time by cutting the messages in smaller packets. A classical technique is pipelining. Suppose we cut the message into $\frac{L}{B}$ packets of length B and send the packets one after each other on a shortest-paths spanning tree. The broadcasting time is $ecc(r)(\beta + B\tau)$ for the first packet to reach a farthest node, plus $(\frac{L}{B} - 1)(\beta + \beta\tau)$ for the other $\frac{L}{B} - 1$ packets following the first one to reach a farthest node. Hence the total time is $(\beta + B\tau)(ecc(r) + \frac{L}{B} - 1)$. This time is optimized by choosing $B_{\min} = \sqrt{L\beta/(ecc(r) - 1)\tau}$, its value being $(\sqrt{L\tau} + \sqrt{(ecc(r) - 1)\beta})^2$. If $L\tau$ is large compared with β, we have now a time of the order $L\tau$. Thus we have earned a factor of $ecc(r)$, but are still far from the lower bound.

Another technique consists of finding p spanning trees rooted at r and pairwise arc disjoint. We cut the message into blocks, each of size $\frac{L}{p}$, and send each block on a different spanning tree. Suppose the maximum depth of the spanning trees is h, then the broadcasting time is $h(\beta + \frac{L}{p}\tau)$. This technique has been used for networks such as hypercubes [20], folded hypercubes [18], star graphs, and k-ary hypercubes [25]. See [12] for a survey. Moreover, we can use the following theorem of graph theory by Edmonds [10] (see Lovász [24] for a short proof).

THEOREM 3.2 (Edmonds). *The maximum number of pairwise arc-disjoint spanning trees rooted at a vertex r is equal to $c_G(r)$.*

We can use these $c_G(r)$ spanning trees to obtain a broadcasting time $h(\beta + (L/c_G(r))\tau)$. Here again, if $L\tau \gg \beta$, we have a time of order $(hL/c_G(r))\tau$ saving a factor of $c_G(r)$ when h is close to $ecc(r)$.

Finally, we can combine the two techniques of pipelining and arc-disjoint spanning trees. With packets of size B, we obtain a time $h(\beta + B\tau)$ so that the first packet reaches a farthest node in the deepest spanning tree, plus $(L/(c_G(r)B) - 1)(\beta + B\tau)$ for the other $L/(c_G(r)/B) - 1$

packets following the first one to reach this farthest node. Hence the total time is $(\beta + B\tau)(h + L/(c_G(r)B) - 1)$. An optimal choice of the size B of the packets gives a broadcasting time $(\sqrt{L\tau/c_G(r)} + \sqrt{(h-1)\beta})^2$.

THEOREM 3.3. *If the maximum depth of $c_G(r)$ arc disjoint spanning trees rooted at a vertex r is h, then there exists a protocol of broadcasting from r whose time is* $(\sqrt{L\tau/c_G(r)} + \sqrt{(h-1)\beta})^2$.

If $L\tau \gg \beta$, we have a time of order $L\tau/c_G(r)$, matching the lower bound. If β and $L\tau$ are of the same order of magnitude, we have to balance the effects of the number of spanning trees and the depth of these trees, respectively. Ideally we want to find many spanning trees ($c_G(r)$ if possible) of maximum depth as small as possible. In the case of regular classical networks, $c_G(r) = \lambda = d$ and $ecc(r) = D$, and the best we can hope to find is d arc-disjoint spanning trees with a depth of at most $D + 1$. Our aim will be to construct arc-disjoint spanning trees of the de Bruijn networks. Note that for the two-dimensional torus, this problem has been completely solved [26].

Let us finish this section with some remarks.

1. Instead of using arc-disjoint spanning trees, we could use time-disjoint diffusion trees (meaning that at a given time, one arc is used in at most one diffusion tree). However, these trees are more difficult to find and do not allow pipelining.
2. There exists algorithms for finding arc-disjoint spanning trees of any given digraph, but the complexities of these algorithms are high, and there is no good bound on the depth of the trees obtained. So it is interesting to give explicit constructions.
3. It is possible to do the same study for the *half duplex* communication mode. See [14], [22], [27] for results on edge-disjoint spanning trees of an undirected graph.

4. de Bruijn digraphs. The de Bruijn digraph $B(d, D)$ [9] is a digraph in which the vertices are the words of length D on an alphabet of d letters (for instance $\{0, \ldots, d-1\}$). There is an arc from a vertex $x = (x_1, \ldots, x_D)$ to a vertex y if the $D-1$ first letters of y are equal to the $D-1$ last letters of x. That is, there is an arc from $x = (x_1, \ldots, x_D)$ to all the vertices $(x_2, \ldots, x_D, \alpha)$, $\alpha \in \{0, \ldots, d-1)\}$. Therefore, each node of $B(d, D)$ has an out and in degree d and it is easy to check that the diameter is D, and for every vertex r of $B(d, D)$, $c_G(r) = \lambda = d - 1$. The number of vertices is $n = d^D$. Figure 1 shows $B(2, D)$ with $D = 1, 2, 3$.

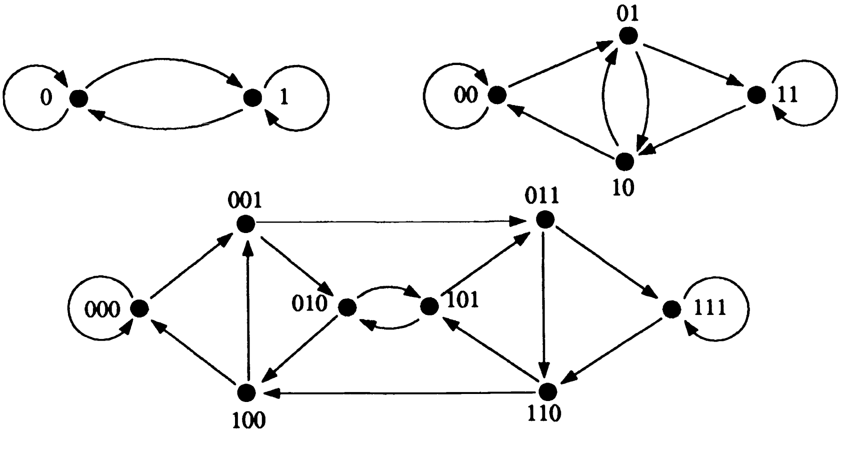


FIG. 1. $B(2, 1)$, $B(2, 2)$, *and* $B(2, 3)$.

One can note that, for a fixed degree d, the number of vertices of these digraphs is of the same asymptotic order as the directed Moore bound (which is equal to $d^D + d^{D-1} + \cdots + d^2 + d + 1$). Another important remark is that the degree and the diameter of these digraphs are independent and adjustable parameters.

The corresponding undirected graphs $UB(d, D)$ are built by omitting the direction of the arcs.

Remark. If we do not consider the loops, all the vertices $(\alpha, \alpha, \ldots, \alpha, \alpha)$ of $B(d, D)$ have in- and out-degree $d - 1$. Many solutions are proposed in the literature [2] to modify the de Bruijn graphs to make them regular, but we do not consider them in this paper. Indeed, links can be added to nodes of degree strictly less than the maximum degree to communicate with external devices such as front-end processors and peripherals.

In the following section, we use the notations $x = (x_1, \ldots, x_D)$ for a vertex of $B(d, D)$ and $e = [x_1, \ldots, x_D, \alpha]$ for the arc $(x_1, \ldots, x_D) \rightarrowtail (x_1, \ldots, x_D, \alpha)$. The capital letter W will represent any sequence of letters, and $|W|$ the length of this sequence.

5. Arc-disjoint spanning trees of the de Bruijn digraph. The main purpose of this section is to construct for a given vertex r of a de Bruijn digraph a set of $c_G(r) = d - 1$ arc-disjoint spanning trees all rooted at this vertex.

We will extensively use the *shortest-paths spanning tree* (or breadth first search tree). The shortest directed path from a vertex x to a vertex y is obtained by determining the longest sequence, common to the end of x and to the beginning of y. Suppose that this longest sequence is of length $D - h$ and of the form $z_1, \ldots, z_{D-h}$ and let $x = (x_1, \ldots, x_h, z_1, \ldots, z_{D-h})$ and $y = (z_1, \ldots, z_{D-h}, y_1, \ldots, y_h)$. Then the distance between x and y is h, and the unique shortest path between x and y contains successively the vertices

$$(1) \qquad \begin{aligned} x^0 &= x \\ x^1 &= (x_2, \ldots, x_h, z_1, \ldots, z_{D-h}, y_1) \\ &\ldots \\ x^i &= (x_{i+1}, \ldots, x_h, z_1, \ldots, z_{D-h}, y_1, \ldots, y_i),\ 1 < i < h \\ &\ldots \\ x^h &= y. \end{aligned}$$

Figure 2 shows the two shortest-paths spanning trees rooted respectively at (000) and (111) in $B(2, 3)$. One can see that they are arc disjoint.

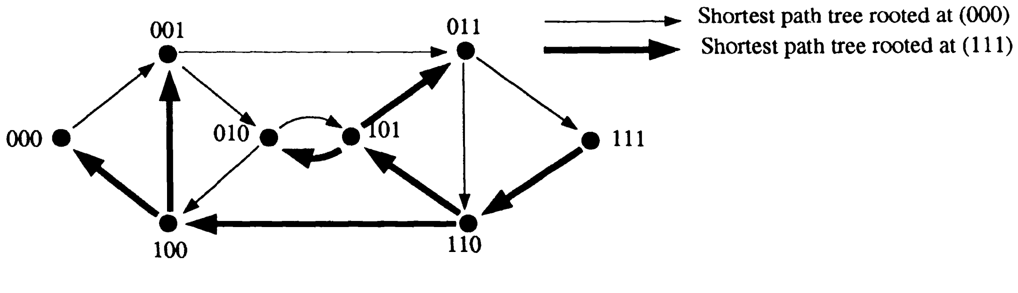

Shortest path tree rooted at (000)
Shortest path tree rooted at (111)
001
011
000
010
101
111
100
110

FIG. 2. *Two shortest-paths spanning trees in* $B(2, 3)$.

PROPOSITION 5.1. *The d shortest-paths spanning trees of the de Bruijn digraph, rooted respectively at the vertices $(\alpha, \ldots, \alpha)$, $\alpha = 0, \ldots, d-1$, are pairwise arc disjoint.*

Proof. Two arcs of different trees are of the form $[\alpha, \ldots, \alpha, W_1]$ and $[\beta, \ldots, \beta, W_2]$, with $|W_i| \leq D, i = 1, 2$. Hence, they are disjoint if $\alpha \neq \beta$. □

Thus, assuming that each of these nodes is connected to an external device, it is possible to load data from the front end to the parallel multiprocessor using d arc-disjoint spanning trees, with a bandwidth d/τ. This is a main point: de Bruijn graphs ensure high bandwidth between any external device and the multiprocessor. We show in the following how to ensure high bandwidth communications within the network.

It is more difficult to find arc-disjoint spanning trees of small depth all rooted at the same vertex in a de Bruijn digraph. It might be worth noting at this point that Imase et al. [19] have built $d-1$ arc-disjoint *paths* of length $\leq D+1$ between any two nodes to study the vulnerability of the de Bruijn digraph. Unfortunately, their construction cannot be used to find arc-disjoint spanning trees rooted at the same vertex.

Finally, let us note that we cannot construct more than $d-1$ arc-disjoint spanning trees rooted at a given vertex due to the presence of loops at the vertices $(\alpha, \ldots, \alpha), \alpha = 0, \ldots, d-1 (c_G(r) = \lambda = d-1)$.

First we present a simple construction of $d-1$ arc-disjoint spanning trees rooted at a given vertex of the form $(\alpha, \alpha, \ldots, \alpha, \alpha)$ with $\alpha \in \{0, \ldots, d-1\}$. Next we generalize this construction.

PROPOSITION 5.2. For all $\alpha \in \{0, \ldots, d-1\}$, *there exist $d-1$ arc-disjoint spanning trees of optimal depth $D+1$ rooted at vertex $(\alpha, \alpha, \ldots, \alpha, \alpha)$ of $B(d, D)$.*

Proof. Let $\alpha \in \{0, \ldots, d-1\}$. With each $\nu \in \{0, \ldots, d-1\}$, $\nu \neq \alpha$ we associate a tree T_ν consisting of the arc from $(\alpha, \ldots, \alpha)$ to $(\alpha, \ldots, \alpha, \nu)$ and the shortest-paths spanning tree rooted at $(\alpha, \ldots, \alpha, \nu)$ from which the vertex $(\alpha, \ldots, \alpha)$ which is a leaf, is deleted. Let $\nu \neq \mu$, and consider any arc $e_\nu = [\alpha, \ldots, \alpha, \nu, W_1]$ from T_ν and any arc $e_\mu = [\alpha, \ldots, \alpha, \mu, W_2]$ from T_μ where $0 \leq |W_i| \leq D, i = 1, 2$. The letters α, ν, μ are all distinct, thus $e_\nu \neq e_\mu$ and T_ν and T_μ are arc disjoint.

It is easy to verify that the depth of these trees is $D+1$. This is optimal since there is only one path of length $\leq D$ from $(\alpha, \ldots, \alpha)$ to any vertex $(x_1, \ldots, x_D), x_1, \neq \alpha$. □

THEOREM 5.3. *For any vertex u of $B(d, D)$, there exist $d-1$ arc-disjoint spanning trees rooted at u of depth at most $D+2k+1$, where k is the length of the second longest subsequence of identical letters in u, that is, of depth at most $D + 2\lfloor \frac{D}{2} \rfloor + 1$.*

Proof. Let $u = (u_1, \ldots, u_D)$ be a given node of $B(d, D)$. We construct $d-1$ arc-disjoint spanning trees rooted at u in two steps. First we will use the shortest-paths trees rooted at vertices at distance $k+1$ from u where k depends on the form of u. Then, if an arc is common to two trees, we will remove this arc from one tree and replace it with another arc that does not appear elsewhere. During all these replacements we will keep the tree structure. We give an example of two arc-disjoint spanning trees of $B(3, 2)$ rooted at $(1, 0)$.

1. *Definition of k and α:* Let us consider a longest subsequence of u consisting of identical letters, and let α be this letter. Let k be the length of the second longest subsequence of identical letters. For example, if $u = (0, 0, \ldots, 0, 0)$, then $\alpha = 0$ and $k = 0$; if $u = (0, 0, 0, 1, 1, 1)$, then $\alpha = 0$ or 1 and $k = 3$; if $u = (0, 0, 0, 1, 1, 2, 2)$, then $\alpha = 0$ and $k = 2$; if $u = (0, 1, 0, 1, 0, 1)$, then $\alpha = 0$ or 1 and $k = 1$. For $u = (1, 0)$, $k = 1$ and we choose $\alpha = 0$.

2. *Definition of T_ν, $\nu \neq \alpha$:* For all $\nu \in \{0, \ldots, d-1\}$, $\nu \neq \alpha$, we build a tree T_ν. We start with the shortest-path P_ν between $u = (u_1, \ldots, u_D)$ and the node $(u_{k+2}, \ldots, u_D, \nu, \ldots, \nu)$ where the letter ν is repeated $k+1$ times, and we add the shortest-paths tree rooted at

$(u_{k+2}, \ldots, u_D, \nu, \ldots, \nu)$. Then we delete from it all the vertices appearing in P_ν (except $(u_{k+2}, \ldots, u_D, \nu, \ldots, \nu)$) and the subtrees rooted at them, and we reattach these subtrees directly to P_ν. Thus we have obtained a spanning tree T_ν rooted at u. In $B(3, 2)$, from $u = (1, 0)$ and our choice of $\alpha = 0$, there are two paths P_1 and P_2:

$$P_1 = (1, 0) \longrightarrow (0, 1) \longrightarrow (1, 1),$$
(2)
$$P_2 = (1, 0) \longrightarrow (0, 2) \longrightarrow (2, 2),$$

and the two trees T_1 and T_2 are shown in Fig. 3 (T_1 is the black tree, and T_2 is the grey tree). They are not disjoint, since $(1, 0) \longrightarrow (0, 2)$ belongs to both trees.

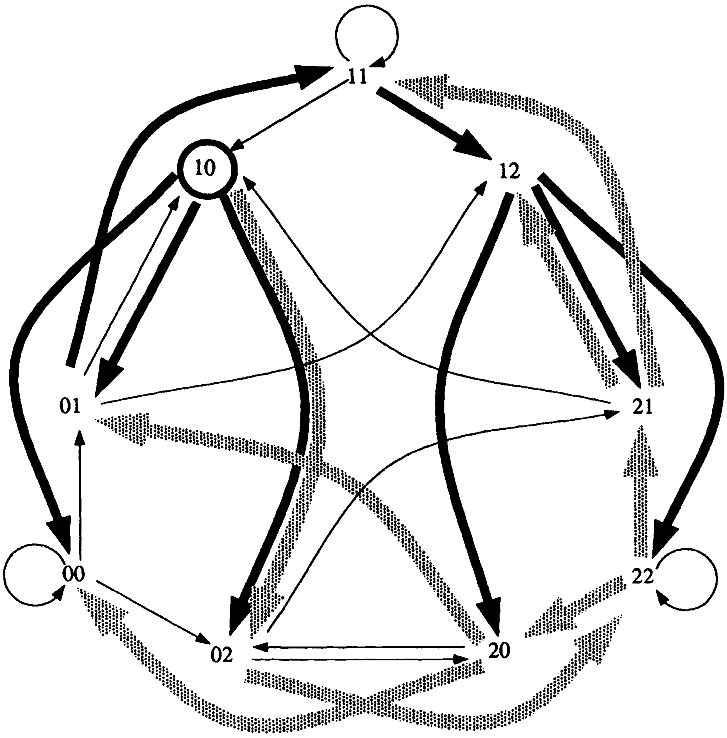


FIG. 3. *Construction of two arc-disjoint spanning trees rooted at* (10) *in* $B(3, 2)$.

3. *Definition of* T'_ν, $\nu \neq \alpha$: An arc of T_ν is of the form

(a) $$e_\nu = \left[u_i, \ldots, u_D, \underbrace{\overbrace{\nu, \ldots, \nu}^{\leq k+1}}_{\geq 1} \right] \text{ if it belongs to } P_\nu,$$

(b) $$\text{otherwise } e_\nu = \left[u_i, \ldots, u_D, \overbrace{\nu, \ldots, \nu}^{k+1}, W \right] \text{ where } W \text{ is non-empty and } i \leq D,$$

(c) $$\text{or } e_\nu = \left[\underbrace{\overbrace{\nu, \ldots, \nu}^{\leq k+1}}_{\geq 1}, W \right] \text{ where } W \text{ is non-empty.}$$

Let $\mu \neq \nu$ and $e_\nu = e_\mu$ where e_ν and e_μ are arcs of T_ν and T_μ, respectively. Since $\mu \neq \nu$, e_ν and e_μ cannot be both of type (a) or both of type (c). Suppose e_ν is of type (a) or (b), and e_μ of type (b). Then $e_\nu = [u_i, \dots, u_D, \nu, \dots, \nu, W]$ where W may be empty (and there may be less than $k+1$ consecutive letters ν), and

$$e_\mu = \left[u_j, \dots, u_D, \overbrace{\mu, \dots, \mu}^{k+1}, W' \right]$$

where W' is non-empty and $j \leq D$. If W is empty (type (a)), then, since $\mu \neq \nu$, there are $k+1$ consecutive letters μ in the subword $u_i, \dots, u_D$, contradicting the definition of k as $\mu \neq \alpha$. If W is non-empty, then we can suppose without loss of generality that $j > i (i \neq j$ since $\mu \neq \nu)$. Thus, as before, there would be $k+1$ consecutive letters μ in the subword $u_i, \dots, u_D$, a contradiction. Note that the same argument shows that e_μ cannot be of type (c) with $k+1$ letters μ at the beginning.

Thus exactly one arc is of type (c) with not more than k identical letters at the beginning. Suppose without loss of generality that it is $e_\mu = [\mu, \dots, \mu, W]$ (W is non-empty). Moreover, this arc is also of type (a) or (b) in T_ν, that is, of the form $[u_i, \dots, u_D, \nu, \dots, \nu, W']$ (W' may be empty), $\nu \neq \mu$. We denote this kind of arc e^ν_μ. Such an arc satisfies

$$\begin{aligned} e^\nu_\mu &= [\mu, \dots, \mu, W],\ W \text{ non-empty} \\ &= [u_i, \dots, u_D, \nu, \dots, \nu, W'],\ W' \text{ can be empty.} \end{aligned} \tag{3}$$

We will replace this arc in T_μ by the arc f^ν_μ which is obtained from e^ν_μ by replacing the first occurrence of μ by α.

Let us call T'_μ the new graph obtained from T_μ by doing all the possible replacements.

In the example in Fig. 3, the arc $[1, 0, 2]$ belongs to both T_1 and T_2. This arc, denoted e^2_1, is of type (c) since it can be written $[1, W]$ (an arc of T_1) and of type (a) since it can be written $[u_1, u_2, 2]$ (an arc of T_2). This arc is replaced in T_1 by the arc $f^2_1 = [\alpha, W] = [0, 0, 2]$ to build T'_1. The other tree is not modified: $T'_2 = T_2$. One can check on Fig. 3 that T'_1 and T'_2 are arc disjoint.

4. *The T'_μ, $\mu \neq \alpha$ are arc disjoint*: The arc f^ν_μ comes from the replacement of $e_\mu = e_\nu$ and so can be written

$$\begin{aligned} f^\nu_\mu &= [\alpha, \mu, \dots, \mu, W],\ W \text{ non-empty} \\ &= [\alpha, u_{i+1}, \dots, u_d, \nu, \dots, \nu, W'],\ W' \text{ can be empty.} \end{aligned} \tag{4}$$

First, let us show that $f^\nu_\mu \neq f^{\nu'}_{\mu'}$ for all $\mu \neq \mu'$. Necessarily, $\nu \neq \nu'$, otherwise there would be two arcs entering to the same node in T_ν, namely $e^\nu_\mu = [\mu X]$ and $e^\nu_{\mu'} = [\mu' X]$ where $X = (u_{i+1}, \dots, u_D, \nu, \dots, \nu, W')$. We have

$$\left\{ \begin{aligned} f^\nu_\mu &= [\alpha, u_{i+1}, \dots\dots, u_D, \nu, \dots, \nu, W], \\ f^{\nu'}_{\mu'} &= [\alpha, u_{j+1}, \dots\dots, u_D, \nu', \dots, \nu', W'], \end{aligned} \right. \tag{5}$$

with at least one ν and one ν'. Without loss of generality, we can suppose that $j > i$ (therefore $i + 1 \leq D$), W' is non-empty, otherwise W would also be empty and this would imply $\nu = \nu'$. Therefore $f^{\nu'}_{\mu'}$ contains a subsequence of $k+1$ letters ν'. Thus $\alpha, u_{i+1}, \dots, u_D$ contains also a subsequence of $k+1$ letters ν' contradicting the definition of k (since $\alpha \neq \nu'$).

Finally, f_μ^ν cannot be an arc of T_γ, $\gamma \neq \mu$. Indeed, assume that $f_\mu^\nu = e_\gamma$:

$$f_\mu^\nu = [\alpha, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W] \tag{6}$$

and

$$e_\gamma = \begin{cases} [u_j, \dots, u_D, \gamma, \dots, \gamma] \text{ (type (a))} \\ \text{or} \\ [u_j, \dots, u_D, \gamma, \dots, \gamma, W'] \text{ (type (b))} \\ \text{or} \\ [\gamma, \dots, \gamma, W''] \text{ (type (c)).} \end{cases} \tag{7}$$

The arc e_γ cannot be of type (c) because $\gamma \neq \alpha$. If $\gamma \neq \nu$, a similar argument shows that there is a subsequence of $k + 1$ letters γ or ν in $u = (u_1, \dots, u_D)$ contradicting the definition of k. But $\gamma = \nu$ implies that the node $(u_{i+1}, \dots, u_D, \nu, \dots, \nu, W)$ is reached in T_ν by the arc $[\alpha, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W]$. This is impossible since it is reached in T_ν by the arc $e_\nu = [u_i, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W] = e_\mu^\nu$, and $u_i \neq \alpha$ since $e_\mu^\nu = [\mu, \dots]$.

5. *The T'_μ, $\mu \neq \alpha$ are connected*: It suffices to show that the path in T'_μ from the root to the tail of the new arc $[\alpha, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W']$ is identical to the corresponding path in T_μ. Suppose that some arc of this path has been modified. Then it must be of the form $[\mu, \dots, \mu, \alpha, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W'']$ (because the path must reach the tail of $[\alpha, u_{i+1}, \dots, u_D, \nu, \dots, \nu, W']$), and it must also be of the form $[u_j, \dots, u_D, \gamma, \dots, \gamma, W''']$ for some $\gamma \neq \mu$.

If $\gamma \neq \nu$, then a similar argument as before shows that there is a subsequence of $k + 1$ letters γ or ν in $u = (u_1, \dots, u_D)$ contradicting the definition of k. So $\gamma = \nu$, but it again implies a contradiction with the definition of k by looking carefully at the positions of u_i in the two forms of the arc and noting that $\nu \neq \alpha$, $u_i \neq \alpha$, and $u_i \neq \nu$ (recall that in fact $u_i = \mu$).

6. *The depth of the T'_μ is at most $D + 2\lfloor \frac{D}{2} \rfloor + 1$*: The path from the root to any vertex in T_μ is of length at most $k + 1 + D$. In the replacement process, we might attach at most one (by step 5) subtree of depth at most $k - 1$. Indeed, the root of this subtree is the head of an arc e_μ^ν, which is (by step 3) of the type (c) but cannot contain more than k letters μ. Altogether the total depth is at most $k + 1 + D + 1 + k - 1 = D + 2k + 1$. We always have $k \leq \lfloor \frac{D}{2} \rfloor$. □

The proof of Theorem 5.3 is constructive and gives a method to construct arc-disjoint spanning trees of $B(d, D)$. With the notation of the proof, first find α and k. Next build the $d - 1$ shortest-paths trees rooted at the $d - 1$ vertices $(u_{k+2}, \dots, u_D, \nu, \dots, \nu)$, $\nu \neq \alpha$ (where the letter ν is repeated $k + 1$ times). Finally, correct the trees following the described rule. Concerning the depth, note that $k = \lfloor \frac{D}{2} \rfloor$ is a worst case that occurs with a small probability. In general, k is much smaller than $\lfloor \frac{D}{2} \rfloor$, hence the depth of the arc-disjoint spanning trees is much smaller than $D + 2\lfloor \frac{D}{2} \rfloor + 1$. For instance, if the root is $u = (\alpha, \alpha, \dots, \alpha, \alpha)$, our construction is the one given in Proposition 5.2 and we have a depth of $D + 1$.

COROLLARY 5.4. *Let r be any node of $B(d, D)$. There exists a protocol of broadcasting from r whose time is at most $(\sqrt{L\tau/(d - 1)} + \sqrt{2DB}\,)^2$.*

For a large message, this time is of the same order as the lower bound given by Proposition 3.1, i.e., $b_{B(d,D)}(r) \geq \max(D\beta, L\tau/(d - 1)\tau)$.

6. Gossiping in networks. First we consider a given digraph $G = (V, E)$ of diameter D and minimum in-degree $d^-_{\min} = \min_{u \in V} d^-(u)$, and next we study the particular case of the de Bruijn digraph. Gossiping is broadcasting from all the nodes. We assume that all the messages are of the same length L.

6.1. General lower bounds. Since during any gossiping all the nodes must perform a broadcast, the total start-up time is at least $\max_{r\in V} ecc(r)\beta = D\beta$. Let u be any node of V and S be a set of vertices not containing u. All the $|S|$ messages initiated by the vertices in S must reach u through $m^+(S, V - S)$ communication links, hence the total propagation time is at least

$$\max_{u\in V}\max_{S\neq\emptyset\,|\,u\notin S}\frac{|S|L}{m^+(S, V - S)}\tau. \tag{8}$$

For instance, choosing $S = V - \{u\}$ the total propagation time is at least

$$\max_{u\in V}((n-1)L/d^-(u))\tau = ((n-1)L/d^-_{\min})\tau.$$

PROPOSITION 6.1. *In a digraph G of minimum in-degree $d^-_{\min}$ and diameter D, the gossiping time is at least $g_G \geq \max(D\beta, ((n-1)L/d^-_{\min})\tau)$.*

Note that we cannot obtain a lower bound by adding the two lower bounds (start-up and propagation time) [13], [31].

According to the above reasoning and particularly to (8), a good gossiping algorithm might proceed in D steps, ensuring that for any set of vertices S maximizing the ratio $|S|/m^+(S, V - S)$, the messages crossing from S to $V - S$ use the $m^+(S, V - S)$ links with a well-balanced load of the messages on the links. Johnsson and Ho [20] and MacKenzie and Seidel [25] show that this is possible for hypercubes and star graphs, respectively. We will show that this is also possible for the de Bruijn digraphs.

6.2. Gossiping in digraphs. We give below a simple greedy algorithm that appears to reach an optimal propagation time in the de Bruijn digraph. More details concerning the algorithm and its extensions can be found in [3], [4], [23]. We define the receiving phase as follows: receive while data arrive through any link and while all the links have not transmitted at least one message. With this convention, we describe following gossiping algorithm.

ALGORITHM (the algorithm is given for a processor u)
Step $i (1 \leq i \leq D)$:

i. Form a message called *New* consisting of all the messages u has not already sent (at step 1, *New* will consist of the message of u itself). Send *New* to all the out-neighbors of u.
ii. Wait until you receive all the messages from your in-neighbors.

Note that to ensure that the algorithm works, u has to send a message at each phase i. If u has not received any new messages at phase $i - 1$, it can send an empty message or a special one.

The algorithm applied to the de Bruijn digraph is shown in Fig. 4. (The messages are numbered with the name of the processor in decimal arithmetic.)

The following lemma shows that the algorithm realizes the total exchange of the messages in D steps and will enable us to compute an upper bound of the gossiping time.

LEMMA 6.2. *At step i, each processor p forwards all the messages from all processors p' such that there exists a shortest path of length $i - 1$ from p' to p $(d(p', p) = i - 1)$.*

Proof. Let us call P_i the property "*at step i, each processor p forwards all the messages from all processors p' such that $d(p', p) = i - 1$.*" We proceed by induction. P_1 is true. Assume that P_j is true for all $j < i$. During step $i - 1$, each processor q sends to all its

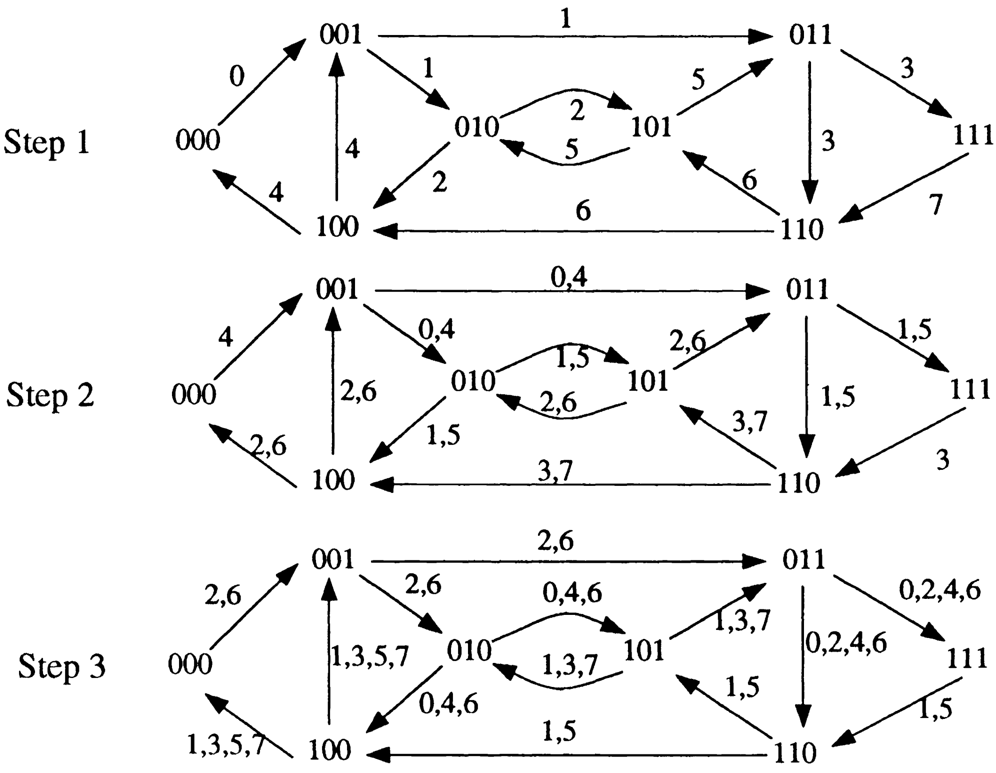


FIG. 4. *Gossiping in* $B(2,3)$.

neighbors all the messages from processors p' such that $d(p', q) = i - 2$ by the induction hypothesis. Thus all the messages from processors p' such that $d(p', p) \leq i - 1$ have been received by p at the end of step $i - 1$. During step i, p forwards messages from processors p' such that $d(p', p) = i - 1$ since it had already forwarded those from processors p' such that $d(p', p) < i - 1$ by hypothesis. Thus P_i is true. □

Let $d^-_{\max}$ be the maximum in-degree of the considered network. From a processor p, there are at most $(d^-_{\max})^i$ processors p' such that $d(p', p) = i$; thus the maximum time of step i of the greedy algorithm is less than $\beta + (d^-_{\max})^{i-1}L\tau$ assuming that all messages are of same length L. Hence:

$$\begin{aligned} T_{Greedy} &\leq \sum_{i=1}^{D} \left(\beta + (d^-_{\max})^{i-1}L\tau\right) \\ &= D\beta + \frac{(d^-_{\max})^D - 1}{d^-_{\max} - 1}L\tau. \end{aligned} \tag{9}$$

THEOREM 6.3. *In any digraph G of diameter D and maximum in-degree $d^-_{\max}$, there exists a protocol of gossiping whose running time is at most*

$$D\beta + \frac{(d^-_{\max})^D - 1}{d^-_{\max} - 1}L\tau.$$

For short messages, the greedy algorithm runs in $\Theta(D\beta)$, which is optimal. But depending on the considered network, the upper bound of the greedy algorithm cost can be far from the lower bound of the time to gossip. However, we will show in the next section that, for the de Bruijn digraph, this upper bound is of the same order as the lower bound for large messages.

6.3. Gossiping in de Bruijn digraphs. For a de Bruijn digraph $B(d, D)$, $d^-_{\min} = d - 1$, thus $g_G \geq (n - 1/d - 1)L\tau$. Moreover, $d^-_{\max} = d$ and $n = d^D$, thus the greedy algorithm applied to the de Bruijn digraph has a complexity $T_{Greedy} \leq D\beta + (n - 1/d - 1)L\tau$.

COROLLARY 6.4. *In the de Bruijn digraph $B(d, D)$, there exists a protocol of gossiping whose running time is at most $D\beta + (n - 1/d - 1)L\tau$.*

For large messages, this protocol is of the same order as the lower bound. Moreover, for any length of message, it is at most two times slower than an optimal algorithm since the lower bound given by Proposition 6.1 is $\max(D\beta, ((n - 1)L/d - 1)\tau)$. Note that Fig. 4 shows that there may exist a dissymmetry on the load of the arcs during each step of the greedy algorithm. Moreover, note that there are small redundancies in the transmission of the data. Hence, it may be possible to decrease the global cost by a small amount.

7. Case of undirected graphs. Designers prefer to construct networks based on undirected graphs. Indeed, layouts of mono- and bidirectional links are of the same complexity. If there is a link between u and v, then a message can be sent directly from u to v and from v to u. Recall that if only one of these messages can be sent at any given time, it is a *half duplex* mode of communication. Otherwise it is a *full duplex*. Here we are interested in the full duplex communication mode and so it is better to think in terms of symmetric digraphs. We can therefore apply the results above. We will now examine only what is happening for the undirected de Bruijn graph $UB(d, D)$.

A node $u = (u_1, \ldots, u_D) \in UB(d, D)$ is linked with $(u_2, \ldots, u_D, \alpha)$ and $(\alpha, u_1, \ldots, u_{D-1})$. A very important point is that we do not remove any of the double bidirectional links between nodes. For instance, (010) is linked twice in $UB(2, 3)$ with (101) by two bidirectional links. The maximum degree of $UB(d, D)$ is $2d$, but its minimum degree is $2d - 2$. We will use $UB^*(d, D)$ to denote the symmetric digraph obtained from $UB(d, D)$.

7.1. Broadcasting. Consider the set T_v of $d - 1$ arc-disjoint trees as constructed in §5. They use the arcs of $UB^*(d, D)$ in only one direction. Moreover, we can consider another family S_v of $d - 1$ arc-disjoint spanning trees by using right shifts instead of left shifts. Therefore, we have constructed a family of $2d - 2$ arc-disjoint spanning trees of $UB^*(d, D)$ of depth h at most $2D + 1$. Thus using Theorem 3.3, we have:

COROLLARY 7.1. *In any symmetric de Bruijn graph $UB^*(d, D)$ there exists a protocol whose broadcasting time is at most $(\sqrt{L\tau/2d - 2} + \sqrt{2D\tau})^2$.*

For large messages, this time is of the same order as the lower bound given by Proposition 3.1, i.e., $b_{UB^*(d,D)}(r) \geq \max(D\beta, (L/2d - 2)\tau)$.

7.2. Gossiping. We easily deduce from the greedy algorithm of §6 a greedy algorithm for $UB^*(d, D)$. Each message is divided into two parts. One part is diffused using the original gossiping (with left shifts), whereas the second part is diffused performing a similar gossiping with right shifts. The complexity is then:

$$\begin{aligned} T_{Greedy} &\leq \sum_{i=1}^{D} \left(\beta + d^{i-1}\frac{L}{2}\tau\right) \\ &= D\beta + \frac{d^D - 1}{d - 1}\frac{L}{2}\tau \qquad (10) \\ &= D\beta + \frac{n - 1}{2d - 2}L\tau. \end{aligned}$$

COROLLARY 7.2. *In the de Bruijn graph $UB^*(d, D)$, there exists protocol of gossiping whose running time is at most $D\beta + \frac{n-1}{2d-2}L\tau$.*

As for $B(d, D)$, for large messages this protocol is of the same order as the lower bound, and for any length of messages it is at most two times slower than an optimal algorithm since the lower bound given by Proposition 6.1 is $g_{UB^*(d,D)} \geq \max(D\beta, \frac{(n-1)L}{2d-2}\tau)$.

8. Conclusion. The de Bruijn digraphs (or undirected graphs) are a good family for performing broadcasting or gossiping. Our broadcasting and gossiping algorithms have propagation times that reach the optimal order for large messages. To do this, we have constructed a family of arc-disjoint spanning trees. These trees have a small depth and are easy to construct when the root is a vertex of the form $(\alpha, \dots, \alpha)$. This confirms the interest of using these vertices as gates with the outside world. It will be interesting to improve the depth of the spanning trees constructed in general. It remains to study the same problem for other families such as Kautz digraphs or generalized de Bruijn or Kautz graphs. Finally, the general problem of bounding the maximum depth of arc-disjoint spanning trees in general graphs is interesting in itself and, as far as we know, algorithms for constructing arc-disjoint spanning trees of minimum depth have not been yet proposed (this problem has been proved to be NP-complete by Noga Alon; see sketch of his proof in [1]).

Acknowledgments. We are grateful to Joseph Peters and Dominique Sotteau for many helpful discussions and remarks.

REFERENCES

[1] J.-C. BERMOND AND P. FRAIGNIAUD, *Broadcasting and NP-completeness*, Graph Theory Notes of New York, XXII (1992), pp. 8–14.

[2] J.-C. BERMOND, N. HOMOBONO, AND C. PEYRAT, *Large fault-tolerant interconnection networks*, Graphs and Combinatorics, 5 (1989), pp. 107–123.

[3] J.-C. BERMOND AND J.-C. KONIG, *General and efficient decentralised protocols* 2, in International Workshop on Parallel and Distributed Algorithms, Bonas (1988), Elsevier-North Holland, Amsterdam, 1989, pp. 199–210.

[4] J.- C. BERMOND, J.-C. KONIG, AND M. RAYNAL, *General and efficient decentralised consensus protocols*, Second International Workshop on Distributed Algorithms, Amsterdam, 1987, Lecture Notes in Comput. Sci. 312, New York, Springer-Verlag, 1988, pp. 41–56.

[5] J.-C. BERMOND AND C. PEYRAT, *Broadcasting in de Bruijn networks*, in Proceedings of the 19th Southeastern Conference on Combinatorics, Graph Theory and Computing, Congr. Numer., 66 (1988), pp. 267–282.

[6] ———, *de Bruijn and Kautz networks: a competitor for the hypercube?*, in Hypercube and Distributed Computers, F. ANDRE AND J. VERJUS, eds., Elsevier-North Holland, Amsterdam, 1989, pp. 279–294.

[7] R. CYPHER, *Theoretical aspects of VLSI pin limitations*, TR 89-02-01, Dept. of Comput. Sci., University of Washington, Seattle, 1989.

[8] W. DALLY AND C. SEITZ, *Deadlock-free message routing in multiprocessor interconnection networks*, IEEE Trans. Comput., c-36 (1987), pp. 547–553.

[9] N. DE BRUIJN, *A combinatorial problem*, Koninklijke Nederlandse Academie van Wetenschappen Proc., A49 (1946), pp. 758–764.

[10] J. EDMONDS, *Edge-disjoint branchings, combinatorial algorithms*, in Combinatorial Algorithms. R. Rustin, ed., Algorithmics Press, New York, 1972, pp. 91–96.

[11] P. FRAIGNIAUD, *Performance analysis of broadcasting in hypercubes with restricted communication capabilities*, J. Parallel Dist. Comput., 16 (1992), pp. 15–26.

[12] P. FRAIGNIAUD AND E. LAZARD, *Methods and problems of communication in usual networks*, Discrete Appl. Math. (special issue on broadcasting), to appear.

[13] P. FRAIGNIAUD, S. MIGUET, AND Y. ROBERT, *Scattering on a ring of processors*, Parallel Comput., 13 (1990), pp. 377–383.

[14] D. GUSFIELD, *Connectivity and edge-disjoint spanning trees*, Inform. Proc. Lett., 16 (1983), pp. 87–89.

[15] S. T. HEDETNIEMI, S. HEDETNIEMI, AND A. LIESTMAN, *A survey of gossiping and broadcasting in communication networks*, Networks, 18 (1986), pp. 319–349.

[16] M. Heydemann, J. Opatrny, and D. Sotteau, *Broadcasting and spanning trees in de Bruijn and Kautz networks*, Discrete Appl. Math. (to appear).

[17] W. Hillis, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

[18] C. Ho, *Full bandwidth communications on folded hypercubes*, in Proceedings International Conference on Parallel Processing, 1990.

[19] M. Imase, T. Soneoka, and K. Okada, *Fault-tolerant processor interconnection networks*, Systems Comput. Japan, 17 (1986), pp. 21–30.

[20] S. Johnsson and C.-T. Ho, *Optimum broadcasting and personalized communication in hypercubes*, IEEE Trans. Comput., 38 (1989), pp. 1249–1268.

[21] P. Kermani and L. Kleinrock, *Virtual cut-through: a new computer communication switching technique*, Computers Networks, 3 (1979), pp. 267–286.

[22] S. Kundu, *Bounds on the number of disjoint spanning trees*, J. Combin. Theory, 17 (1974), pp. 199–203.

[23] T. Lakshman and W. Wei, *Efficient decentralized consensus protocol using specially structured communication graphs*, Technical Report Bellcore (submitted to IEEE Trans. Comput.), 1990.

[24] L. Lovász, *On two minimax theorems in graph theory*, J. Combin. Theory, Ser B, 21 (1976), pp. 96–103.

[25] D. MacKenzie and S. Seidel, *Broadcasting on three multiprocessor interconnection topologies*, CS-TR-89-01, Michigan Technical University, Houghton, 1989.

[26] P. Michallon, D. Trystram, and G. Villard, *Optimal broadcasting algorithms on torus*, Technical report RR872-I-0192, LMC, INPG, Grenoble, 1992.

[27] C. St. J. A. Nash-Williams, *Edge-disjoint spanning trees of finite graphs*, J. London Math. Sec., 36 (1961), pp. 445–450.

[28] Y. Saad and M. Schultz, *Data communication in parallel architectures*, Parallel Comput., 11 (1989), pp. 131–150.

[29] M. Samatham and D. Pradhan, *The de Bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI*, IEEE Trans. Comput., 38 (1989), pp. 567–581.

[30] S. Seidel, *Circuit-switched vs. store andforward solutions to symmetric communication problems*, Proceedings of the 4th Conference on Hypercube Concurrent Computers and Application, 1989.

[31] Q. Stout and B. Wagar, *Intensive hypercube communication, prearranged communication in link-bound machines*, J. Parallel Dist. Comput., 10 (1990), pp. 167–181.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 227–246, April 1994

A LINEAR-TIME ALGORITHM FOR THE HOMOTOPIC ROUTING PROBLEM IN GRID GRAPHS*

MICHAEL KAUFMANN† AND KURT MEHLHORN‡

Abstract. The paper considers the problem of finding edge-disjoint paths between pairs of vertices in a finite grid graph. The homotopy class for each path to be routed is prespecified. A very fast algorithm that guarantees to find a solution for any solvable homotopic routing problem is given.

Key words. algorithms, homotopic routing, edge-disjoint paths, VLSI-theory

AMS subject classifications. 68Q25, 68U05

1. Introduction. We give a linear time algorithm for the homotopic routing problem in grid graphs.

Problem: Homotopic Routing Problem in Grid Graphs (HRP)
Input: A grid graph R and nets $q_1, \dots, q_k$.
Output: Pairwise edge-disjoint grid paths $p_1, \dots, p_k$ such that p_i is homotopic to q_i, $1 \le i \le k$, or an indication that no such paths exist. □

The *planar rectangular grid consists* of vertices $\{(x, y); x, y \in \mathbb{Z}\}$ and edges $\{((x, y), (x', y')); |x - x'| + |y - y'| = 1\}$. A *grid graph* $R = (V, E)$ is a finite subgraph of the planar rectangular grid. We call a bounded face F of R trivial if it has exactly four vertices on its boundary and nontrivial otherwise. We muse M to denote the set of nontrivial bounded faces together with the unbounded face F_{ext} and $\mathcal{O}$ to denote the union of the interiors of the faces in M. A nontrivial face is also called a hole.

A path p is a continuous function $p : [0, 1] \to \mathbb{R}^2 - \mathcal{O}$. A path p is called a *net* if $\{p(0), p(1)\} \subseteq V \cap \partial\mathcal{O}$ where $\partial\mathcal{O}$ is the boundary of $\mathcal{O}$. Two paths p and q are *homotopic*, denoted $p \sim q$, if there is a continuous function $F : [0, 1] \times [0, 1] \to \mathbb{R}^2 - \mathcal{O}$ such that $F(0, x) = p(x)$ and $F(1, x) = q(x)$ for all $x, 0 \le x \le 1$, and $F(t, 0) = p(0)$ and $F(t, 1) = p(1)$ for all $t, 0 \le t \le 1$. A path p is called a *grid path* if $p(x)$ belongs to R for all x.

Figure 1 gives an example of an HRP. For the algorithmic treatment, we assume that the nets $q_1, \dots, q_k$ are grid paths and use n to denote the number of vertices of R plus the total number of edges in the paths q_i. The integer n is called the size of the HRP. We use $\mathcal{N}$ to denote the set $\{q_1, \dots, q_k\}$ of nets.

THEOREM 1. *Let $P = (R, \mathcal{N})$ be an even bounded HRP of size n.*

(a) *P is solvable if and only if* $\text{fcap}(X) \ge 0$ *for every cut X.*

(b) *In time $O(n)$ one can decide whether P has a solution and also construct a solution if it does.* □

Part (a) of this theorem was shown in [KM2] and later extended by [Sh]. The paper [KM2] also presents an $O(n^2)$ algorithm. In the present paper we give a new algorithm with linear running time.

A *cut* C is a simple path in $\mathbb{R}^2 - \mathcal{O} - V$ with its endpoints in $\partial\mathcal{O}$. The *capacity* $\text{cap}(C)$ of a cut C is the number of intersections with edges of R. If C is a cut and p is a path then $\text{cross}(p, C)$ is the number of intersections with edges p and C and $\text{mincross}(p, C) =$

*Received by the editors February 13, 1989; accepted for publication (in revised form) June 10, 1992. This work was supported by the DFG, Sonderforschungsbereich 124, Teilprojekt B2, VLSI Entwurf und Parallelität.

†Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany.

‡MPI für Informatik, Universität des Saarlandes, Im Stadtwald 15, D-66123-Saarbrücken 11, Germany.

min{cross (q, D); $q \sim p$, $D \sim C$}. Finally, the *density* dens(C) of cut C is defined by

$$\text{dens}\,(C) = \sum_{p \in \mathcal{N}} \text{mincross}(p, C)$$

and the *free capacity* fcap (C) is given by

$$\text{fcap}(C) = \text{cap}(C) - \text{dens}(C).$$

A cut C is *saturated* if fcap$(C) = 0$ and *oversaturated* if fcap$(C) < 0$. An HRP is *even* if fcap(C) is even for every cut C.

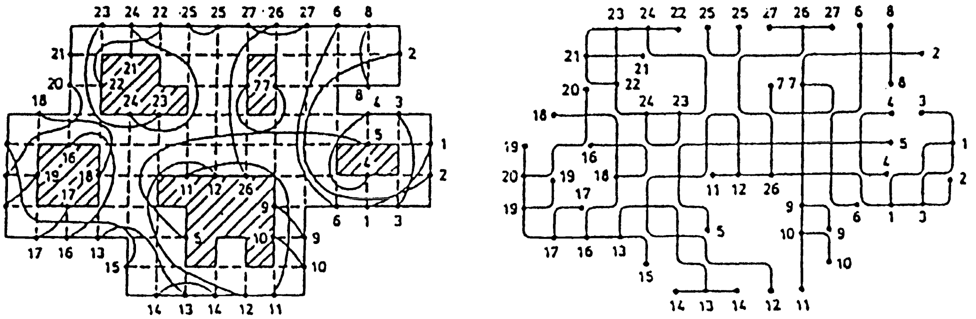


FIG. 1.

Let v be a vertex in R. We denote the degree of v by deg(v) and the number of nets having v as endpoint by ter(v). An HRP is *bounded*, if deg(v) + ter$(v) \leq 4$ for all vertices v, and *weakly bounded* if deg$(v) = 4$ implies ter$(v) = 0$.

There are many previous papers on finding edge-disjoint paths in grid graphs and general planar graphs, e.g., [PL], [F], [MP], [NSS], [KM1], [KM2], [BM], [K], [Sh]. The present paper extends the work in [KM2]. We refer the reader to [KM2] for a discussion of the relationship between VLSI-design and homotopic routing problems.

This paper is organized as follows. In §2 we describe the algorithm and in §3 we prove its correctness. The algorithm is similar in spirit to the algorithm in [KM2], but differs in many details. In particular, its correctness does not follow from [KM2]. In §4 we then describe the linear time implementation of the algorithm. Weinelt [W] has implemented the algorithm; Fig. 1 has been produced by his program.

2. The algorithm. In this section we describe an algorithm for the homotopic routing problem in grid graphs. Recall that we are given a grid graph R and nets $p_1, \dots, p_k$ and that our goal is to shift the nets $p_1, \dots, p_k$ into pairwise edge-disjoint grid paths. Our algorithm works iteratively. In each iteration we consider an edge e of R and decide whether to use it for some net and if so for which one. If e is to be used for net p then we choose suitable nets p^1 and p^2 with $p \sim p^1 e p^2$ and replace net p by the three nets p^1, e, and p^2. The edge e is then used to route the net e in the obvious way. Thus, each iteration discards one edge of R and hence there are $O(n)$ iterations. In §4 we show how to implement the algorithm such that each iteration takes amortized time $O(1)$.

For the algorithm we need some further concepts.

DEFINITION 1. *For a path p the canonical representation* can(p) *is the shortest path homotopic to p.*

Note that can(p) is composed of straight-line segments.

DEFINITION 2. (a) *A path p is called a prefix of path q if there is a monotone function $t : [0, 1] \to [0, 1]$ with $t(0) = 0$ such that $p(x) = q(t(x))$ for all x.*

(b) *The reversal p^{-1} of path p is defined by $p^{-1}(x) = p(1 - x)$ for $0 \leq x \leq 1$.*

(c) *If p and q are paths, then $p \approx q$ if either $p \sim q$ or $p^{-1} \sim q$.*

(d) *For a path p, we define* source$(p) = p(0)$ *and* target$(p) = p(1)$.

(e) *For a point v, $x(v)$ and $y(v)$ denote the x- and y-coordinate of point v, respectively.*

DEFINITION 3. *Let p and q be nontrivial paths with the same source s and let s lie on the boundary of a unique hole F. Then p is said to be* right *of q, if either*

- can(p) = can(q) *or*
- can(p) *is a proper prefix of* can(q) *and there is a hole to the right of* can(q) *at point* target(can(p)) *or*
- can(q) *is a proper prefix of* can(p) *and there is a hole to the left of* can(p) *at point* target(can(q)) *or*
- can(p) *and* can(q) *have no nontrivial common prefix and for every sufficiently small circle K around s there is a counterclockwise scan of K intersecting first ∂F, then* can(p), *then* can(q), *and finally again ∂F or*
- can(p) *and* can(q) *have a maximal common nontrivial prefix r, i.e.,* can$(p) = rp^1$, can$(q) = rq^1$ *and p^1 and q^1 have no nontrivial common prefix, and for every sufficiently small circle K around* target(r) *there is a counterclockwise scan of K intersecting first r, then p^1, and finally q^1.*

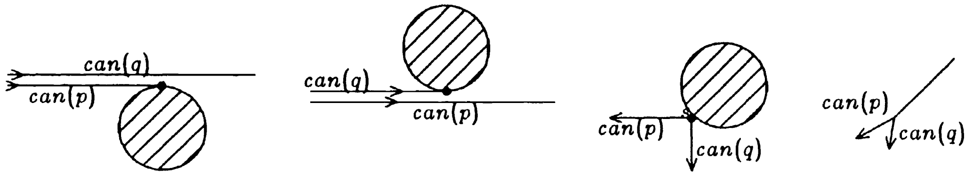


FIG. 2. *An illustration of the four cases in Definition* 3.

Figure 2 illustrates the various cases of Definition 3. The relation *left* is defined analogously. Both relations are clearly transitive.

DEFINITION 4. (a) *A path p is* x-monotone *if x*(target(p)) $\leq x$(source(p)) *and for every vertical line L the intersection $p \cap L$ is a segment, i.e., x-monotone paths travel from right to left.*

(b) *The* monotone prefix pref(p) *of a net p is the maximal x-monotone prefix of* can(p).

(c) *Let p and q be nets with a common source s and let s lie on the boundary of a unique hole. Then p is* quasi-right *of q if either* pref(q) *is trivial or* pref(p) *and* pref(q) *are nontrivial and* pref(p) *is right of* pref(q).

Remark. The ordering right played an important role in the algorithm of [KM2]. In this paper we use the ordering quasi-right instead. This is one of the sources for the improved running time.

DEFINITION 5. *A cut C is called* vertical *if it is a vertical straight-line segment.*

The algorithm is given as Program 1. In this program, *top* always denotes the largest y-coordinate of any vertex of R, *Top* is the subgraph spanned by the vertices with y-coordinate equal to *top*, and a *segment* is a connected component of *Top*. An endpoint of a segment is called *exposed* if it is not the terminal of any net. The endpoints of a segment are also called *corners* of the segment. Finally, *Rim* is the closed region above the horizontal line $y = top - 1$. In the formulation of Program 1 we assumed that the lowest numbered case for which the precondition is satisfied is executed, i.e., if e.g., case 4.3 is taken, then cases 1, 2, 3, 4.1, and 4.2 do not apply. In cases 4.2 and 5.1 the concept of rightmost decomposition is used, which we now define.

DEFINITION 6. *Let $e = (b, a)$ with b the left neighbor of a be an edge in row Top and let X be the vertical cut through e. Let C be the set of nets p with* $\text{mincross}(p, X) > 0$. *For $p \in C$ an* admissible *decomposition is a triple (p^1, e, p^2) such that $p \approx p^1ep^2$ and* $\text{mincross}(p^1, X) + \text{mincross}(p^2, X) = \text{mincross}(p, X) - 1$. *A triple (p^1, e, p^2) is called a* rightmost *decomposition with respect to X if it is an admissible decomposition of a net $p \in C$ and p^2 is quasi-right of q^2 for any admissible decomposition (q^1, e, q^2).*

```
while   $R \neq \emptyset$
do
  case 1:  $\exists$ cut $X$ intersecting an edge $e$ in Top with $\text{cap}(X) = 1$ (*operation 1*)
           let $X$ be such a cut, let $e$ be the intersected edge, and let $p$ be the unique
           net with $\text{mincross}(p, X) = 1$;
           let $p \approx p^1ep^2$ with $\text{mincross}(p^1, X) = 0$ for $i = 1, 2$;
           replace $p$ by $p^1$ and $p^2$ and discard $e$;
  case 2:  $\exists$ segment $S$ consisting of a single vertex $b$(*operation 2*)
           let $(b, c)$ be the vertical edge incident to $b$ and let $p$ be the unique net
           incident to $b$; let $p \approx (b, c)p^1$; replace $p$ by $p^1$ and discard $(b, c)$;
  case 3:  $\exists$ non-trivial segment $S$ with its right corner $b$ not exposed
           (*operation 3*)
           let $p$ and $q$ be the two nets incident to $b$ with $p$ quasi-right of $q$ and
           let $e = (b, a)$ be the horizontal edge incident to $b$;
           let $p \approx ep^1$; replace $p$ by $p^1$ and discard $e$;
  case 4:  $\exists$ non-trivial segment $S$ with its right corner $b$ exposed and its left
           corner not exposed;
           case 4.1:  $\exists$ net $p$ with $\text{can}(p) \subseteq S$(*operation 4.1*)
                      delete net $p$ and discard all edges in $\text{can}(p)$
           case 4.2:  the vertical cut $X$ through the horizontal edge $e =$
                      $(b, a)$ is saturated (*operation 4.2*)
                      let $(p^1, e, p^2)$ be the rightmost decomposition of any
                      net which crosses $X$; replace $p$ by $p^1$ and discard all
                      edges in $\text{can}(ep^2)$
           case 4.3:  let $p \approx (a, b)p^1$ be a net incident to the left neighbor
                      $a$ of $b$;
                      replace $p$ by $p^1$ and discard edge $(a, b)$
                      (*operation 4.3*)
  case 5:  $\exists$ nontrivial segment $S$ with both corners exposed
           case 5.1:  $\exists$ saturated vertical cut through an edge of $S$ (*operation 5.1*)
                      let $X$ be the shortest saturated vertical cut through an
                      edge $e = (b, a)$ of $S$; here $b$ is the right neighbor of $a$;
                      let $(p^1, e, p^2)$ be the admissible decomposition of a net
                      with respect to $X$ such that either can $(p^1) \subseteq Rim$ or
                      the decomposition is rightmost;
                      replace $p$ by $p^1$ and $p^2$ and discard $e$
           case 5.2:  let $a$ and $b$ be the left and right corner of
                      $S$(*operation 5.2*);
                      add nets $p$ and $q$ with $p(\lambda) = q(\lambda) = \lambda a + (1 - \lambda)b$
                      for $0 \leq \lambda \leq 1$
od
```

PROGRAM 1.

3. Proof of correctness. This section is divided into two subsections. In the first subsection we collect some useful facts about cuts and in the second subsection we argue the correctness of the routing algorithm.

3.1. Some facts about cuts.

DEFINITION 7. *A cut $C : [0, 1] \to \mathbb{R} - \mathcal{O} - V$ is called* straight *if the function C is linear and it is called* almost-straight *if it is either straight or if the line segment connecting $C(0)$ and $C(1)$ is contained in the boundary of a nontrivial face, the functions $C|[0, \frac{1}{2}]$ and $C|[\frac{1}{2}, 1]$ are linear, there is no vertex of R contained in the interior of the triangle $C(0)C(\frac{1}{2})C(1)$, and no edge of R is intersected more than once by C. A cut C is called* Manhattan *if it consists of horizontal and vertical straight-line segments. For a cut C,* Manhattan(C) *is a Manhattan cut intersecting the same edges as C.*

DEFINITION 8. *A tuple $(p_1, \ldots, p_k)$ is called* straight-line decomposition *of p if* can(p) $= p_1 \cdots p_k$ *and each p_i is a maximal straight-line segment contained in* can(p). *Each p_i is called a straight-line piece of p.*

LEMMA 1. (a) *Let P be weakly bounded. Then the cut condition holds if it holds for all almost-straight cuts.*

(b) *Let P be bounded. Then* fcap(C) > 0 *for every cut C that is almost-straight but not straight. Also, the cut condition holds if it holds for straight cuts.*

(c) *Let p be a net and C a cut. Then* cross(p, C) $\equiv$ mincross(p, C) mod 2.

(d) *Let $(p_1, \ldots, p_k)$ be the straight-line decomposition of p and let C be a straight cut. Then* mincross(p, C) $= \sum_{i=1}^{k}$ cross(p_i, C).

(e) *Let P be a bounded problem satisfying the cut condition and let C be a saturated Manhattan path in P. Then consecutive turns of C are in alternate directions.*

(f) *Let P be bounded. If there is an oversaturated straight cut in P, then there is either an oversaturated horizontal or vertical cut in P or an oversaturated straight cut intersecting only edges incident to vertices of degree* 4.

Proof. (a) This is the content of §3 of [Sh].

(b) Let C be almost-straight but not straight. Then the line segment connecting $C(0)$ and $C(1)$ passes through exactly cap(C) $- 2$ vertices, all of which have degree 3. Hence dens(C) $\leq$ cap(C) $- 2$. The second part now follows from part (a).

(c) This is Lemma 7 of [KM2].

(d) Since $p \sim$ can(p) $= p_1 p_2 \cdots p_k$, we have mincross(p, C) $\leq \sum_i$ cross(p_i, C). Let us assume mincross(p, C) $< \sum_i$ cross(p_i, C). Then there are (x_1, λ_1), (x_2, λ_2) such that $x_i \neq x_2$, $\lambda_1 \neq \lambda_2$, and can(p)$|[x_1, x_2] \sim C|[\lambda_1, \lambda_2]$. Since can($p$) is a shortest path homotopic to p and since C is a straight-line segment, we conclude that can(p)(x_1) and can(p)(x_2) must lie on the same segment p_i. So p_i is a subpath of C and hence C passes through a vertex of R, a contradiction.

(e) This is Lemma 12a, Claim 1 of [KM2].

(f) This is Lemma 12b of [KM2]. Note that this lemma state in the terminology of that paper that there is either an oversaturated 0-bend or an oversaturated 1-bend cut connecting two concave corners. The latter gives rise to a straight cut intersecting only edges incident to vertices of degree 4. □

3.2. Correctness of the algorithm. All of the actions in our algorithm fall under the following paradigm. For a certain edge e and a certain net p we choose p^1 and p^2 such that $p \approx p^1 e p^2$, replace p by p^1 and p^2 and discard e; operations 4.1, 4.2, and 5.2 can be viewed as repeated application of this basic action. It is convenient to view the basic step as to consist of two substeps.

substep A: replace p by the three nets p^1, e, and p^2.

substep B: remove edge e and net e.

The further outline of the correctness proof is as follows. We first deal with substep B in Lemma 2, then show in Lemma 3 that substep A generates even problems satisfying the degree constraints postulated in the premise of Lemma 3, and finally show in Lemmas 4 to 10 that operations 1 to 5.2 preserve the cut condition. In the proofs of these lemmas we will frequently use part (a) of Theorem 1, i.e., the present paper does not give an alternative proof of part (a).

LEMMA 2. *Let $P = (R, \mathcal{N})$ be an even routing problem satisfying the cut condition, and let e be an edge in the boundary of $\mathcal{O}$ as well as a net in $\mathcal{N}$. Assume further that* $\text{ter}(v) + \deg(v) \leq 4$ *for all vertices except the endpoints of e and* $\text{ter}(v) + \deg(v) \leq 6$ *for the two endpoints of e. Then removing edge e and net e yields an even, bounded problem satisfying the cut condition.*

Proof. Let us use P' to denote the modified problem. P' is certainly bounded. If there is a cut of capacity one through e, then P' is also even and satisfies the cut condition. So let us assume that there is a trivial face F incident to e in P. Let $R' = R - e, \mathcal{N}' = \mathcal{N} - e$ and let Y' be any straight cut in R'. If Y' is also a cut in P, then $\text{dens}'(Y') = \text{dens}(Y')$ and $\text{cap}'(Y') = \text{cap}(Y')$ where dens′ and cap′ are computed with respect to P'. If, on the other hand, Y' ends in F then we can extend Y' to a cut in P with $\text{cap}(Y) = \text{cap}'(Y') + 1$ and $\text{dens}(Y) = \text{dens}'(Y') + 1$. Thus $\text{fcap}'(Y') = \text{fcap}(Y) \geq 0$. □

LEMMA 3. *Substep* A *generates even problems satisfying the degree constraints postulated in the premise of Lemma* 2.

Proof. The claim about the degree contraints is obvious. For the evenness let C be any cut and let $p \sim p^1 e p^2$. Then

$$\begin{aligned} \text{mincross}(p, C) &= \text{mincross}(p^1 e p^2, C) \\ &= \text{cross}(p^1 e p^2, C) \bmod 2 \qquad \text{by Lemma 1(c)} \\ &= [\text{cross}(p^1, C) + \text{cross}(e, C) + \text{cross}(p^2, C)] \bmod 2 \\ &= [\text{mincross}(p^1, C) + \text{mincross}(e, C) + \text{mincross}(p^2, C)] \bmod 2 \end{aligned}$$

by Lemma 1(c). Thus evenness is preserved. □

LEMMA 4. *Operations* 1 *and* 2 *preserve the cut condition.*

Proof. Obvious. □

LEMMA 5. *Operation* 3 *preserves the cut condition.*

Proof. In operation 3 there are nets p and q incident to the right corner b of a segment with p quasi-right of q. We replace p by e and p^1, where $e = (b, a)$, a is the left neighbor of b, and $p \approx ep^1$. Assume, for the sake of a contradiction, that there is an oversaturated almost-straight cut C in the modified problem. The cut C must go through edge e since otherwise its density is not affected by the action. Also $\text{mincross}(e, C) = 1$ and $\text{mincross}(p^1, C) = \text{mincross}(p, C) \pm 1$. Thus the density of C is increased by at most 2 and hence C must be saturated in the original problem. Thus C must be straight by Lemma 1(b) and $\text{mincross}(p^1, C) = \text{mincross}(p, C) + 1$. Since the original problem has a solution in which then necessarily net q uses edge e, we conclude $\text{mincross}(q^1, C) = \text{mincross}(q, C) - 1$ where $q \approx eq^1$. Let $(p_1, \ldots, p_h)$ and $(q_1, \ldots, q_k)$ be the straight-line decompositions of p and q, respectively. Then p_1 does not intersect C since $p_1 \cdots p_k$ is a shortest path homotopic to p, $\text{mincross}(p, C) = \sum_i \text{cross}(p_i, C)$ by Lemma 1(d), and $\text{mincross}(p^1, C) = \text{mincross}(p, C) + 1$, and q_1 intersects C for analogous reasons.

Case A. $\lfloor x(\text{target}(C)) \rfloor \leq \lfloor x(\text{source}(C)) \rfloor$: Then p_1 is not right of q_1 and hence p is not quasi-right of q.

Case B. $\lfloor x(\text{target}(C))\rfloor > \lfloor x(\text{source}(C))\rfloor$: Consider the cut D indicated in Fig. 3. Then D is saturated, since $\text{mincross}(p, D) = \text{mincross}(p, C) + 1$ and $\text{mincross}(q, D) = \text{mincross}(q, C) - 1$. Also, Manhattan ($D$) makes two consecutive turns into the same direction, a contradiction to Lemma 1(e). □

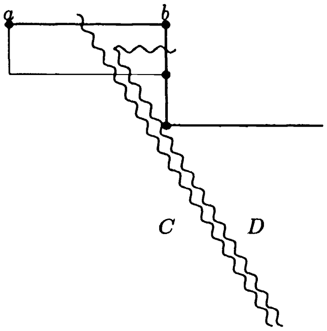


FIG. 3.

LEMMA 6. *Operation* 4.1 *preserves the cut condition. This is even true if the right corner of S is not exposed.*

Proof. In operation 4.1 there is a net p with $\text{can}(p) \subseteq S$. Then $\text{mincross}(p, C) = \text{cross}(\text{can}(p), C)$ for every straight cut C. Hence, deleting the net p and discarding all edges in $\text{can}(p)$ preserves the cut condition. □

LEMMA 7. *Operation* 4.2 *preserves the cut condition.*

Proof. In operation 4.2 the right corner of S is exposed and the left corner of S is not exposed. Furthermore, there is no net p with $\text{can}(p) \subseteq S$ and hence the horizontal cut Y, which cuts off the segment S, is saturated. Note that there are exactly $\text{cap}(Y)$ terminals above segment Y and that each terminal contributes one to the density of Y. Let b be the right corner of S, let $e = (b, a)$ be the horizontal edge incident to b, and let X be the vertical cut through e. In operation 4.2 the cut X is saturated. Let (p^1, e, p^2) be the rightmost decomposition of any net with respect to X.

CLAIM 1. $\text{can}(p^2) \subseteq S$.

Proof. Let $e' = (c, b)$ be the vertical edge incident to b. Consider any solution to the original problem. Then there must be a net, say q, which uses the edges e' and e, i.e., the solution path can be written $q^1e'eq^2$. Furthermore, since there are exactly $\text{cap}(Y)$ terminals in S, contributing one each to the density, we conclude that $\text{mincross}(q^2, Y) = 0$. Thus $\text{can}(q^2) \subseteq S$. Next observe that p^2 is quasi-right of q^2, since (p^1, e, p^2) is a rightmost decomposition with respect to X, and hence $\text{can}(p^2) \subseteq S$. □

Let P' be the problem obtained by replacing p by p^1 and ep^2. It suffices to show that P' satisfies the cut condition, since the deletion of net ep^2 and all edges in $\text{can}(ep^2)$ is covered by Lemma 6. So assume that there is an oversaturated straight cut C in P'. By Lemma 1(f) we may assume that C is either vertical or horizontal or a straight cut intersecting only edges incident to vertices of degree 4. Since $\text{cross}(ep^2, C) \leq 1$, $p^1 \approx ep^2p^{-1}$ and hence $\text{mincross}(p^1, C) \leq \text{cross}(\text{can}(p), C) + \text{cross}(ep^2, C) \leq \text{mincross}(p, C) + 1$, we conclude $\text{fcap}(C) = 0, \text{fcap}'(c) = -2$, $\text{mincross}(ep^2, C) = 1$, and $\text{mincross}(p^1, C) = \text{mincross}(p, C) + 1$. From $\text{mincross}(ep^2, C) = 1$ we conclude that C must intersect an edge of segment S and hence must be vertical.

Since $\text{mincross}(p, X) > 0$, we can write $\text{can}(p) = r^1r^2$ where $x(\text{source}(r^2)) = x(b)$, $\text{cross}(r^2, X) = 1$, and the vertical segment s^1 connecting $\text{source}(r^2)$ to b is contained in R. Then $p^1 \sim r^1s^1$ and hence $\text{mincross}(p^1, C) \leq \text{cross}(r^1s^1, C) = \text{cross}(r^1, C) = \text{cross}(\text{can}(p), C) - 1 = \text{mincross}(p, C) - 1$. Thus $\text{fcap}'(C) = \text{fcap}(C)$ for any vertical cut, a contradiction. □

LEMMA 8. *Operation* 4.3 *preserves the cut condition.*

Proof. The precondition is almost as in operation 4.2; however, the vertical cut X through $e = (b, a)$ is not saturated. Let p be a net which has a as a terminal. Let $p \approx (a, b)p^1$ and replace p by (a, b) and p^1.

Assume that there is an oversaturated straight cut C with respect to the modified problem P'; note that P' is bounded. Then C must intersect the edge (a, b), since otherwise its density is not affected by the action, and $\text{fcap}(C) = 0$. Next observe that P' is bounded and hence by Lemma 1(f) we may assume that C is vertical, i.e., $C = X$. This contradicts $\text{fcap}(C) = 0$. □

LEMMA 9. *Operation* 5.1 *preserves the cut condition.*

Proof. Let X be the shortest saturated vertical cut intersecting an edge in S, let $e = (b, a)$ be the edge in S intersected by X with b being the right neighbor of a, and let $p \approx p^1ep^2$ be a net crossing X such that either $\text{can}(p^1) \subseteq Rim$ or (p^1, e, p^2) is the rightmost decomposition with respect to X. We replace p by p^1, e, and p^2.

The following claim will be useful.

CLAIM 2. *In P there is no saturated nonvertical cut through edge e.*

Proof. Assume that there is such a Manhattan cut C'. By Lemma 1(e) we may assume that C' turns in alternate directions. Assume also that the first segment C_1' of C' is maximal in length. We now distinguish cases. Assume first that C' has only 1 bend or C_1' intersects less that $\text{cap}(X) - 1$ edges.

We move the vertical part of C' intersecting e so as to make the cut C' shorter. Then one of the following situations must arise (see Fig. 4). Either we split the cut at some point into several cuts or we move the vertical part beyond the end of segment S. In the first case one of the parts is a saturated vertical cut intersecting S and being shorter than X; in the second case an oversaturated cut is obtained since the corners of S are exposed. In both cases, we have a contradiction. We conclude that C_1 intersects $\text{cap}(X) - 1$ edges and C' has more than 1 bend; i.e., the situation is as shown in Fig. 5.

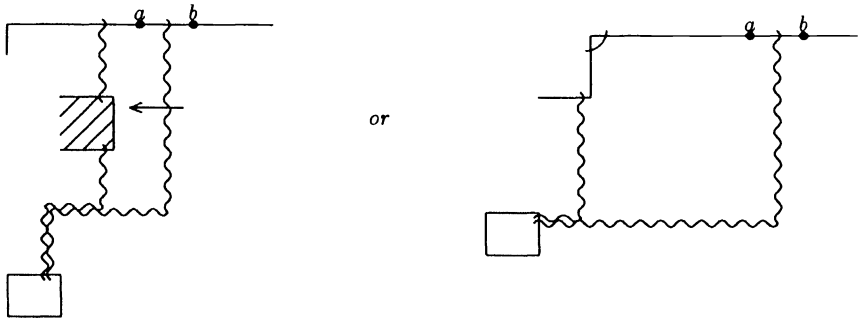


FIG. 4. *Illustrating the first case in the proof of Claim* 2.

We now move the first horizontal part of C' down by one unit, and in this way split C' into several cuts, which are all saturated in P. Hence one of them is oversaturated in P'. But the topmost cut ($= X$) is saturated in P' and the free capacities of all other cuts do not change, since they do not interfere with cut X. □

We now return to the discussion of operation 5.1. Assume that there is an oversaturated cut C with respect to the modified problem P'. Since $\text{mincross}(p, X) > 0$, we can write

$\text{can}(p) = r^1fr^2$ where $x(\text{source}(f)) = x(b), x(\text{target}(f)) = x(a)$, and the straight-line segments s^1 connecting source(f) to b and s^2 connecting a to target(f) are contained in R. Then $p \sim s^1r^1, p^2 \sim r^2s^2$, and $f \sim s^1es^2$. Also, $\text{mincross}(p^1, C)+ \text{mincross}(e, C)+ \text{mincross}(p^2, C) \leq \text{cross}(r^1s^1, C) + \text{cross}(e, C) + \text{cross}(s^2r^2, C) = \text{cross}(r^1fr^2, C) + \text{cross}(s^1es^2, C) - \text{cross}(f, C) \leq \text{mincross}(p, C) + 2$. We conclude that $\text{cross}(s^1es^2, C) = 2$, $\text{fcap}(C) = 0$ and $\text{fcap}'(C) = -2$. Hence C is straight by Lemma 1(b).

Case A. C intersects e: Claim 2 and the fact that $\text{fcap}(C) = 0$ and C is straight imply that C is vertical. Thus $C = X$, a contradiction.

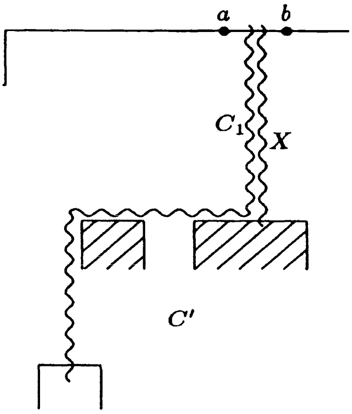


FIG. 5.

Case B. C does not intersect e. Then C must intersect s^1 and s^2 and hence C and X intersect in a single point.

Case B1. $\text{can}(p^1) \subseteq Rim$. Then C must intersect $\text{can}(p^1)$ because otherwise $\text{mincross}(p^1e, C)+\text{mincross}(p^2, C) = \text{mincross}(p^2, C) \leq \text{cross}\,(e^{-1}(\text{can}(p^1))^{-1}\text{can}(p), C) \leq \text{cross}(\text{can}(p), C)$, and hence $\text{fcap}'(C) = \text{fcap}(C)$. Let $C' \sim C$ be a Manhattan cut with $\text{cap}(C') = \text{cap}(C)$. Assume first that C' is a straight horizontal cut. Then $C' = Y$, where Y is defined as in Lemma 7, and hence $f\text{cap}(C') \geq 2$, a contradiction. We conclude that C' contains at least one vertical segment. Since $\text{can}(p^1) \subseteq Rim$, C intersects $\text{can}(p^1)$, s^1, and s^2, C is straight, and since the corners of all segments in *Top* are exposed, we may assume without loss of generality (w.l.o.g.) that C' starts with a vertical segment and turns right. Thus if we move that vertical segment by one unit to the left, the capacity of the cut cannot increase. Also, since the corners of all segments are exposed when case 5 applies, the density of the cut cannot decrease and hence the cut stays saturated in P (or even becomes oversaturated in P'). As we move the vertical part further to the left either one of the following three situations must arise (see Fig. 6). Either, we obtain an oversaturated cut in P or a nonvertical saturated cut through e or the cut is split into several cuts at least one of which is saturated in P and oversaturated in P'. In all three cases we have a contradiction; this is obvious in the first case, follows from Claim 2 in the second case, and follows in the third case form the observation that none of the resulting cuts can simultaneously intersect s^1, s^2, and $\text{can}(p^1)$.

Case B2. (p^1, e, p^2) is the rightmost decomposition with respect to X. Consider the cuts Z_1 and Z_2 shown in Fig. 7. Let Z_1' be a shortest path homotopic to Z_1. Z_1' is a polygonal path that bends at the corners of some holes. We split Z_1' at these corners and obtain straight cuts $Z_{11}', \ldots, Z_{1k_1}'$ (see Fig. 8). In a similar way we obtain $Z_{21}', \ldots, Z_{2k_2}'$ from Z_2.

CLAIM 3. $\text{fcap}(X) + \text{fcap}(C) \geq \sum_{i=1}^{k_1} \text{fcap}(Z_{1i}') + \sum_{i=1}^{k_2} \text{fcap}(Z_{2i}')$.

Proof. It is clear that $\text{cap}(X) + \text{cap}(C) \geq \sum_{i=1}^{k_1} \text{cap}(Z_{1i}' + \sum_{i=1}^{k_2} \text{cap}(Z_{2i}')$. It therefore

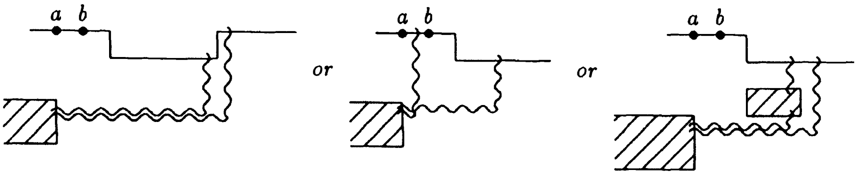


FIG. 6.

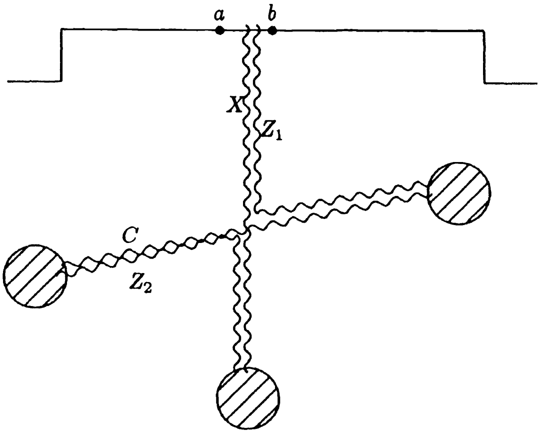


FIG. 7. *X and C produce* Z_1 *and* Z_2.

suffices to show that $\text{cross}(s, X) + \text{cross}(s, C) \leq \sum_{i=1}^{k_1} \text{cross}(s, Z'_{1i}) + \sum_{i=1}^{k_2} \text{cross}(s, Z'_{2i})$ where s is any straight-line piece of a net $q \in \mathcal{N}$. Divide the plane by the lines supporting X and C into four regions as shown in Fig. 9. Then all pieces s contribute the same amount to both sides except those having one endpoint in region D and one endpoint in region F. Such segments contribute 2 to the left hand side and 0 to the right hand side.

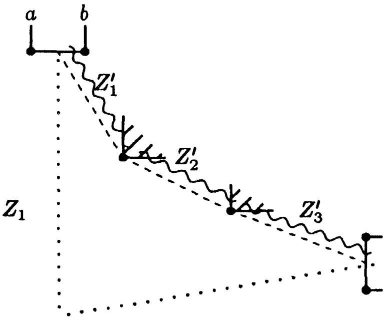


FIG. 8.

We now show that there is no such segment. Assume otherwise. Let q be a net such that $\text{cap}(q) = q'sq''$ contains an elementary piece s, which has its source in F and its target in D. Let $s = s's''s'''$ where $x(\text{source}(s'')) = x(b)$ and $x(\text{target}(s'')) = x(a)$; let l_1 and l_2 be the paths parallel to X and connecting source(s'') with b and a with target(s''), respectively. Then $q \approx q's'l_1el_2s'''q''$, $\text{mincross}(q', X) = \text{mincross}(q's'l_1, X)$ and $\text{mincross}(q'', X) = \text{mincross}(l_2s'''q'', X)$, i.e., $(q's'l_1, e, l_2s'''q'')$ is a decomposition of q with respect to X. Also,

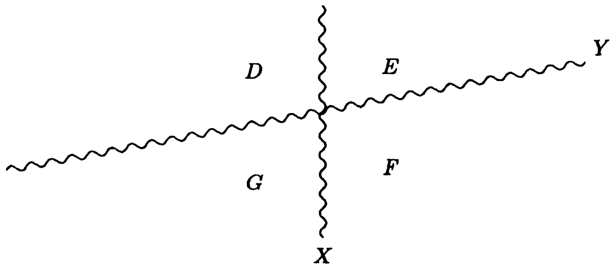


FIG. 9. *The four regions D, E, F, and G.*

$l_2 s''' q''$ is quasi-right of p_2, since the first elementary piece of p_2 ends in G, a contradiction. □

Claim 3 implies that either one of the cuts $Z'_{1i}, 1 \le i \le k_1$, or $Z'_{2i}, 2 \le i \le k_2$, is oversaturated, a contradiction to the assumption that P satisfies the cut condition, or that all of them are saturated. In the latter case, Z'_{11}, a nonvertical straight cut through e, is saturated, a contradiction to Claim 2. The discussion of case B is now completed and Lemma 9 is shown. □

LEMMA 10. *Operation* 5.2 *preserves the cut condition.*

Proof. When operation 5.2 is applied, both corners of S are exposed and there is no saturated vertical cut through any edge of S. Let a and b be the two corners of S. We add the nets p and q with $p(\lambda) = q(\lambda) = a\lambda + (1 - \lambda)b$ for $0 \le \lambda \le 1$. Let P' be the modified problem and assume that there is an oversaturated cut C in P'. Then C must intersect S and C was saturated in P. Since P' is bounded, we may assume that C is vertical by Lemma 1(f), a contradiction. □

4. A linear time implementation. In this section we describe a linear time implementation of our algorithm. In the first part of the section we introduce the required data structures and in the second part we realize the algorithm using these data structures. The first part consists of two interleaved sections: the description of the abstract and the description of the concrete data structure. In the abstract data part we use objects like sequences and sets and operations on these objects and in the concrete part we show how to realize these operations. The concrete part is interleaved with the abstract part.

4.1. The data structure. We first discuss the representation of the routing region. We assume that the vertices and the vertical cuts are numbered 1, 2, 3, ... The numbering of the cuts is such that for each x-coordinate the cuts with the x-coordinate are numbered by consecutive numbers from top to bottom. We identify vertices and cuts with their number. We specify the routing region by storing the neighbors for each vertex.

Concrete. We have four arrays l, r, u, d. For $1 \le v \le |V|, l[v]$ is the (number of the) left neighbor of v; similarly for r, u, and d. We also have two arrays X and Y that give the coordinates of every vertex. □

Let v be any vertex. Then $L(1, v)(L(2, v)$, respectively) is the closest vertex with the same y-coordinate as v to the left of v, which can be reached from v by a straight-line grid path and which is incident to a vertical boundary edge from below and from above, respectively. $R(i, v)$, $U(i, v)$, and $D(i, v), i = 1, 2$ are defined analogously with left replaced by right, above, and below, respectively.

Concrete. We have another four arrays L, R, U, and D. The array L can be initialized in linear time by scanning each row of the routing region from left to right. Similarly, for R, U, and D. □

We turn to the representation of nets next.

DEFINITION 9. (a) *Let p be a net. Then* rightmost p *is a shortest grid path homotopic to p, which is right of any other shortest grid path homotopic to p.* leftmost(p) *is defined analogously.*

(b) *Let p be a net. The monotone decomposition of p is $(p_1, \ldots, p_k)$ where* $\mathrm{can}(p) = p_1 \cdots p_k$, *and p_i is either a maximal vertical segment or a maximal subpath not containing a vertical segment.*

(c) *The representative of p is* $\mathrm{rep}(p) = p'_1 \cdots p'_k$ *where $(p_1, \ldots, p_k)$ is the monotone decomposition of p and*

$$p'_i = \begin{cases} p_i & \text{if } p_i \text{ is vertical,} \\ \text{rightmost } (p_i) & \text{if } x(\text{target}(pi)) < x(\text{source}(p_i)), \\ \text{leftmost}(p_i) & \text{if } x(\text{target}(p_i)) > x(\text{source}(p_i)). \end{cases}$$

(d) *The bend sequence $b_s(p)$ of p is the sequence $v_0, \ldots, v_k$ of vertices such that*

(1) *v_0 and v_k are the terminals of p;*

(2) *$v_1, \ldots, v_{k-1}$ are consecutive bends of* $\mathrm{rep}(p)$*;*

(3) *if $v_0 \notin Rim \cap \partial F_{\mathrm{ext}}$, then v_1 is the first bend of* $\mathrm{rep}(p)$ *and if $v_0 \in Rim \cap \partial F_{\mathrm{ext}}$, then v_1 is the first bend not contained in Rim. Similarly, if $v_r \notin Rim \cap \partial F_{\mathrm{ext}}$, then v_{r-1} is the last bend of* $\mathrm{rep}(p)$ *and if $v_r \in Rim \cap \partial F_{\mathrm{ext}}$, then v_{r-1} is the last bend not contained in Rim.*

Remark. In the bend sequence of a path we suppress the bends contained in Rim, since these bends appear and disappear as we process the top row. It would therefore be very costly to treat them like the other bends. Figure 10 shows the representative of a net and its bend sequence.

In the input, a net is given as a grid path. The representative of a path can be computed in time proportional to its length by walking along the path and looking for shortcuts using the arrays L, R, U, and D.

This can be seen as follows. In a first walk along the path, a shortest path homotopic to the net is computed (the task is here to remove U-turns which are not forced by a hole).

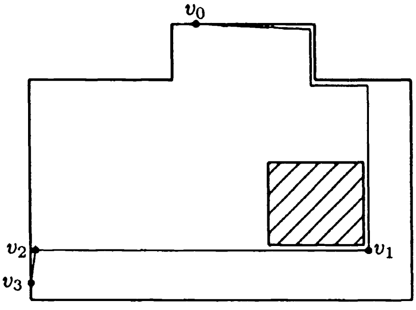


FIG. 10.

In a second walk along the path, the path is decomposed into montone pieces and the pieces are shifted appropriately. All of this takes linear time in the length of the path.

The bend sequence can be computed from the representative by inspecting the initial and final segments of it. Altogether, all representatives and bend sequences can be computed in linear time.

A *crossing* is an intersection between a representative and a vertical cut. Every crossing belongs to two sequences as we describe next.

Let p be a net, let $v_0, \ldots, v_k$ be the bend sequence of p and let $c_1, \ldots, c_m$ be the crossings of rep(p) with vertical cuts in the order in which they occur on rep(p). For $0 \leq i < k$, define the c-sequence of (v_i, v_{i+1}) as the subsequence of crossings that occur between v_i and v_{i+1}.

Let C be any vertical cut. The r-sequence of C is any ordering $c_1, \ldots, c_p$ of the crossings with C satisfying the following three properties: Let $e = (b, a)$, with a the left neighbor of b be the topmost edge intersected by C, and let (p_i^1, e, p_i^2) be an admissible decomposition of net p_i corresponding to crossing c_i, $1 \leq i \leq m$. Then

(1) (*rim property*) There is an integer $k(C) \geq 0$ such that $i \leq k(C)$ implies can $(p_i^1) \subseteq$ *Rim*;

(2) (*ordering property*) If $k(C) < i < j$, then p_i^2 is quasi-right of p_j^2;

(3) (*consistency property*) If D is a vertical cut in the column to the left of C, $k(C) < i < j$ and p_i and p_j also cross D, then $k(D) < i' < j'$ where c_i' is the next crossing of p_i and D and c_j' is the next crossing of p_j and D.

Remark. Initially, we will have $k(C) = 0$, for all C. Then (2) states that the r-sequence of C reflects the ordering quasi-right and (3) states that the orderings for adjacent cuts are consistent with one another. We are not able to maintain (2) and (3) with $k(C) = 0$ for all C. One of the difficulties is case 4.3. In the situation of Fig. 11 the new crossing of p with X would have to be inserted somewhere into the r-sequence of X.

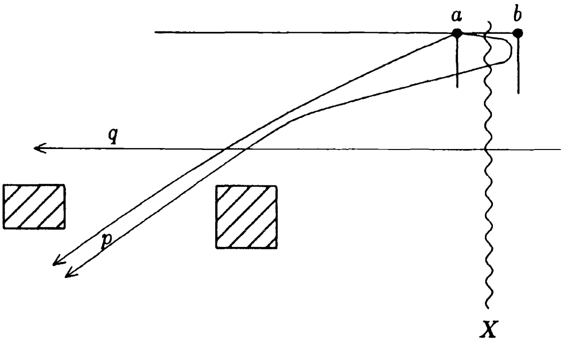


FIG. 11.

We will instead add it to the front of the sequence and increase $k(X)$ by one. In this way (1), (2), and (3) are maintained with small cost. However, the ordering quasi-right in cases 3, 4.2, and 5.1 is now harder to compute. In case 3 we solve this problem by basing the decision on cuts that do not intersect the *Rim* and hence have $k(C) = 0$; in case 4.2 we use the fact that can$(ep^2) \subseteq$ *Top* for the rightmost decomposition (p^1, e, p^2), and in case 5.1 we can always take the first element of the r-sequence. The details are given below.

We call $c_1, \ldots, c_{k(C)}$ the *rim-part* and $c_{k(C)+1}, \ldots, c_m$ the *non-rim-part* of the r-sequence of C.

Concrete. A bend sequence is realized as a doubly linked list (see Fig. 12). The elements of the list are vertices and representatives of c-sequences. Each c-sequence is a doubly linked list of crossings; a crossing is represented by a record to be described below. The representative of a c-sequence points to the first and the last item of the c-sequence. An r-sequence is represented as a singly linked list. There is an array rheads[ ] of list headers for the r-sequences. rhead[i] points to the first element of the r-sequence for cut i. We describe below how this data structure is initialized.

There are several functions that can be applied to crossings and c-sequences. For a crossing c, Findrep(c) returns the representative of the c-sequence to which c belongs, Findcut(c) returns the number of the cut whose r-sequence contains c, Above(c, d) yields true for two crossings c, d belonging to the same r-sequence if c is in front of d in the sequence, Split(c) splits the c-sequence containing the crossing c after c and returns the representatives of the two resulting sequences, Addleft(r, c) and Addright(r, c) add the crossing c to the c-sequence with representative r, Delete(c) deletes the crossing c from the c-sequence containing it, and Access&Split(r, x) splits the c-sequence with representative r after the crossing with x-coordinate x and returns the representatives of the resulting c-sequences.

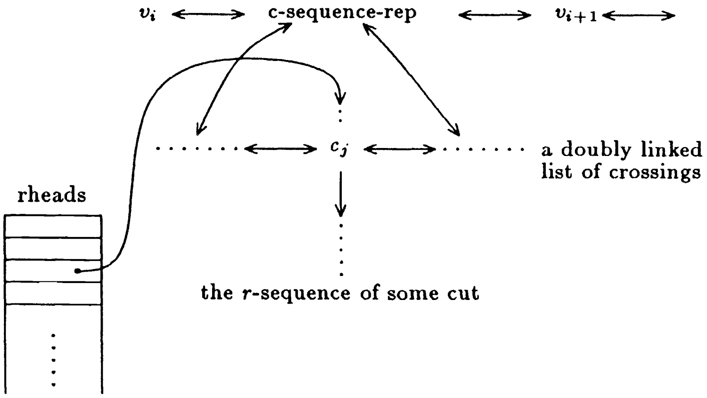

v_i
c-sequence-rep
v_{i+1}
c_j
a doubly linked list of crossings
rheads
the r-sequence of some cut

FIG. 12.

Concrete. We realize a crossing as a record consisting of the following fields: Two pointers *csuc* and *cpred* for the c-sequence; a pointer *rsuc* for the r-sequence; an integer *cutnumber*, which is the number of the cut containing the crossing; an integer *rank*; and various other fields, which are used to realize the other operations. We postulate that the rank field increases along any r-sequence and hence Above takes time $O(1)$. Findcut also takes time $O(1)$ by virtue of the cutnumber field. The operations Findrep, Split, Addleft, Addright, Delete, and Access&Split comprise the data type *splittable list.* It is shown in [Schw] that splittable lists can be implemented such that all operations have amortized cost $O(1)$. The implementation combines the level-linked trees of [HMRT] with the split-find data structures of [GT], [IA].

We are now ready to describe the initialization of the data structure. We number the cuts from left to right and for each x-coordinate from top to bottom. The c-sequences are easily computed in linear time. The r-sequences are computed as follows. We scan the layout region from left to right. When we reach a certain column, the r-sequences for all cuts to the left of the column are already computed, and ranks are assigned to all crossings in these r-sequences. We now show how to compute the r-sequences for all cuts immediately to the right of the column in time proportional to the length of the column (see Fig. 13).

Consider any cut C to the right of the current column. We first divide the set of crossings with C into three parts: the U-, D- and L-parts; the U-part and D-part consist of all crossings with nets that either terminate in the current column on a module that touches the current column from the right and lies above C or below C, respectively, or that have the next crossing with a cut above C or below C, respectively; and the L-part consists of all other crossings. The r-sequence of C can then be obtained by concatenating an appropriate ordering of the U-part with an appropriate ordering of the L-part with an appropriate ordering of the D-part.

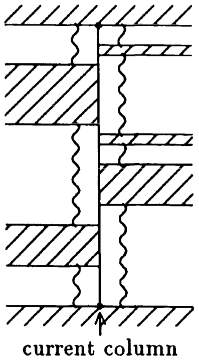


FIG. 13.

We show how to order the L-parts. We first construct a list L consisting of the topmost vertex of the current column, followed by the r-sequence of the first cut to the left of the column, followed by the terminals on the first module touching the current column from the left, followed by the r-sequence of the second cut, ..., and then number the elements of this list in increasing order. We then go through the L-parts of all cuts C and generate for each crossing c with C, say with rep(p), a pair consisting of

- the number of the cut C
- the rank in L of the crossing or terminal adjacent to c in rep(p).

We sort these pairs by bucket sort in time proportional to the length of the column. This gives us the desired ordering of the L-parts, the U- and D-parts can be ordered in a similar way. The ranks are now assigned by numbering the crossings in each q-sequence in increasing order. Altogether we have now shown how to compute the r-sequences of all cuts in linear time.

We finally store for each cut its free capacity (array freecap[ ]). We also have an array satcut[1...] of pointers. The element satcut[i] points to a doubly linked list of all saturated vertical cuts of length i intersecting row *top*. Each saturated cut points to its position on the satcut lists (array satpos[...]). We also keep the nonempty entries of the array *satcut* in a linked list (see Fig. 14). This finishes the description of the data structure.

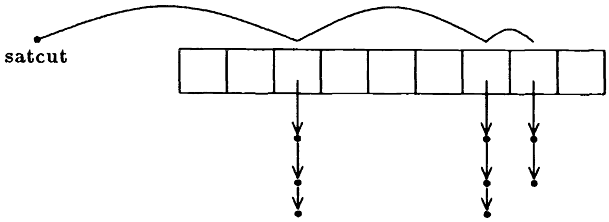


FIG. 14.

4.2. The algorithm. We work through the routing region from top to bottom. Suppose that we are just beginning to process row *top*. Let us also assume that we have the data structure described in §1 available.

Step 1 (*Cuts of capacity one*). A cut of capacity one is necessarily saturated and hence the cuts of capacity one intersecting row *top* are all contained in the list satcut[1]. Let C be any cut in satcut[1], let (a, b) be the edge in row top intersected by C, and let c be the crossing on the r-list of C. Let c be on the c-sequence of bends u and v. Note that we can find u and v in time $O(1)$ using operation Findrep. We split (operation Split) the c-sequence of u and v at the crossing c and introduce a and b as new terminals (see Fig. 15).

At this point we have created two nets, say p_1 and p_2, with terminals in row *top*. If either u or v was a terminal of its net then the bend sequences of p_1 and p_2 are correctly computed by the splitting process. If neither u or v was a terminal, then we have to update the bend sequences of p_1 and p_2 as follows. Follow the bend sequence until a vertex, say w, which is not in Rim is encountered, delete the vertices before w and concatenate the appropriate c-sequences. If l vertices were deleted then $l + 1$ c-sequences have to be concatenated. We concatenate these sequences by adding (operation Add) the items on the second, third, ..., sequence to the first sequence one by one. Note that a crossing can be added at most once to a c-sequence because it belongs to a c-sequence incident to a terminal after the addition. Hence the total cost of additions is linear.

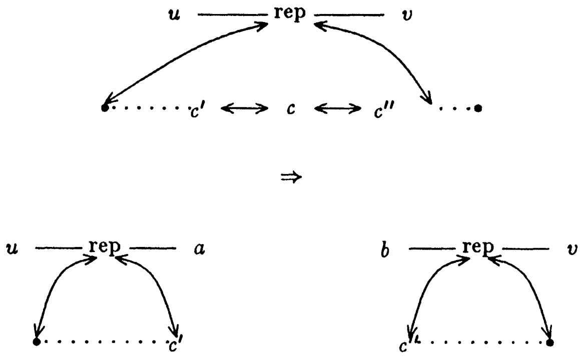


FIG. 15.

Step 2 (*No cuts of capacity one*). All cuts of capacity one are processed. We scan row *top* from left to right and construct four sets of segments. The set RS, LS, and FS, contains all segments where the right corner is not exposed, the right corner is exposed but the left corner is not exposed, and both corners are exposed, respectively. The set TS contains all segments consisting of a single vertex.

We now discuss cases 2 to 5 of the routing algorithm.

Case 2 applies whenever $TS \neq \emptyset$. Let $S \in TS$ be a segment, let b be the single vertex in S, let p be the net incident to b, and let a be the lower neighbor of b. We replace b by a on the bend sequence of p and delete the edge $e = (b, a)$. This takes constant time.

Case 3 applies whenever $TS = \emptyset$ and $RS \neq \emptyset$. Let S be any segment in RS, let b be the right corner of S, let a be the left neighbor of b, and let p and q be the two nets with terminal b. Let v_1 and w_1 be the vertices following b on the bend sequences of p and q, respectively.

Case A. At least one of the nets p and q leaves the column of b to the left. This can be checked in time $O(1)$ by inspecting the first crossing of p and q. We may assume w.l.o.g. that p leaves the column of b to the left. If q does not then p is quasi-right of q. So let us suppose that q also leaves the column of b to the left. If v_1 and w_1 are different or v_1 and w_1 are equal and at least one is a terminal or at least one of the nets makes a left turn at v_1, then it is easy to decide whether p is quasi-right of q using the coordinates of v_1 and w_1. So let us assume that v_1 and w_1 are equal and both nets make a right turn at v_1. In this case both nets enter the bend point v_1 from above. Let c and d be the crossings following v_1 on p and q, respectively, and let C be the cut containing c and d. Then C does not intersect row *top* and hence p is quasi-right of q if c is above d in the r-sequence of C. In either case we have shown that the test whether p is quasi-right of q takes time $O(1)$.

Lėt w.l.o.g. p be quasi-right of q. We replace b by a on the bend sequence of p and delete (operation Delete) the first crossing from the first c-sequence of p. We also replace b by its lower neighbor on the bend sequence of q. Finally, we mark (array mark) the cut, say X, through the edge (a, b) as processed, add it to the set of marked cuts, delete X from its satcut-list, if it is on one, and insert it into the satcut-list of one smaller index. All of this takes time $O(1)$.

Remark. The mark on cut X indicates that X does not start in row *top* anymore. The mark bit will be used in case 5. The set of marked cuts is used to unmark the marked cuts again when row *top* is completely processed.

If p terminates in b, then we remove S from RS and add it to LS or FS whatever is appropriate (a segment is stored as a pair of vertices and hence this is an $O(1)$ decision). If p does not terminate in b, then S stays in RS.

Case B. p and q leave the column of b to the right. Let c and d be the first crossings of p and q and let C and D be the cuts crossed. If $C \neq D$, then it is easy to decide whether p is quasi-right of q. We may assume w.l.o.g. that p is quasi-right of q in this case. If $C = D$, then p is quasi-right of q and q is quasi-right of p. We may assume w.l.o.g. that either c belongs to the rim-part of the r-sequence of C or that both c and d belong to the non-rim-part and d is above c.

We replace b by a on the bend sequence of p, add the crossing of p and X, where X is the vertical cut through edge (a, b), as first element of the rim-part of the r-sequence of X if c belongs to the rim-part of C and as first element of the non-rim-part otherwise (this preserves the ordering property and the consistency property) and add the crossing to the appropriate c-sequence. We also decrease the free capacity of X by two. If X becomes saturated then we add X to the appropriate satcut-list and mark X. The appropriate satcut-list is determined by linear search in time proportional to the length of X. Since every cut becomes saturated at most once, the total cost of adding saturated cuts to satcut-lists is linear.

Case 4 applies whenever $TS = RS = \emptyset$ and $LS \neq \emptyset$. Let S be any segment in LS. The right corner of S is exposed and the left is not.

Case 4.1: There is a net p with $\text{can}(p) \subseteq S$. Let p be the one with leftmost right terminal.

We can find p as follows. We search through the segment starting in the left corner of S. For every vertex v encountered we inspect the bend sequence of the net incident to v and determine in time $O(1)$ whether v qualifies as vertex t. Thus t can be found in time proportional to its distance from the left corner of S.

We route as shown in Fig. 16. All cuts between s and t are marked and moved to the satcut-list of one smaller index, if saturated. Also, all crossings of net p are removed. Also the terminal is changed for each net incident to vertices between s and t, segment S is removed from LS, segment S' is added to LS, and segment S'' is added to FS.

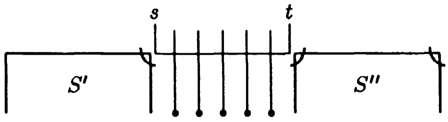


FIG. 16. *Both corners of S'' are exposed and the right corner of S' is exposed.*

Note that the cost of this action is proportional to the length of S' plus the number of edges deleted from the routing region. Since case 4.1 does not again apply to segment S', rather S' is completed by cases 4.2 and 4.3, the total time spent in case 4.1 is linear.

Cases 4.2 and 4.3: There is no net contained in S. Let b be the right corner of S, let a be the left neighbor of b and let X be the vertical cut through (a, b).

Case 4.2: X is saturated. Let p be the quasi-rightmost net across X. We have shown in §3 that p has a terminal s in segment S. We can find p as follows: We start in the right corner of S and walk to the left. For each terminal encountered in the walk we check whether the associated net crosses X by examining its bend sequence. This takes time $O(1)$ per terminal. The first net encountered which crosses X is the desired net p. We route p along the top row (see Fig. 17), route all nets between S and the right corner of S down by one unit, update the data structure as in case 4.1, and put S' on LS.

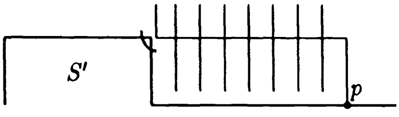


FIG. 17.

Note that case 4.2 or 4.3 applies again to S'. The time required is proportional to the number of edges deleted from the routing region.

Case 4.3: X is not saturated. Let p be the net with terminal a. We route p form a to b and then down by one unit (see Fig. 18).

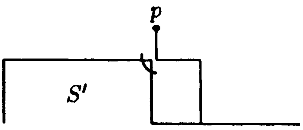


FIG. 18.

If p did not go across X before the action (inspect the first crossing of the net p), then we add a crossing as first element of the rim-part of the r-sequence of X (this preserves the ordering and the consistency property), decrease the free capacity of X by two, and add X to a satcut-list if X became saturated. All of this takes time $O(1)$ if X did not become saturated and time $O(1+$ length of $X)$ otherwise. Thus the total time spent in case 4.3 is linear. Also case 4.2 or 4.3 applies again to segment S'.

Case 5 applies whenever $TS = LS = RS = \emptyset$ and $FS \neq \emptyset$.

We keep a pointer P to the satcut-lists with the following semantics. If the pointer points to list satcut[i] then all cuts in satcut[j], $j < i$, and all cuts preceding the item pointed to in satcut[i] are marked.

We advance P until it points to an unmarked cut. This takes time $O(1)$ per move of P. Note that all cuts added to the satcut-lists during the execution of cases 2, 3, and 4 are marked and hence P never has to be reset. Also the total length of the satcut-list is at most the number of edges in rows top and $top - 1$ and hence the total time spent on advancing P is linear.

Case 5.1: Let X be the saturated cut found, and let (a, b) be the edge in row top intersected by X. Clearly, X is a shortest vertical cut intersecting an edge of the segment containing edge (a, b). Let c be the first crossing on the r-list of C. Let c be on the c-sequence of u and v. Note that we can find u and v in time $O(1)$ using Findrep. Figure 19 shows how the representatives of the nets p^1 and p^2 look like. The bend sequences of p^1 and p^2 can be found as follows.

Let y be the y-coordinate of vertices u and v. Then the bends of p^1 and p^2 are readily determined by inspecting the L- and R-values of all vertices w which lie below a in rows $top, top - 1, \ldots, y$. This takes time $O(1 + top - y)$; since the sum of the lengths of the representatives of p^1 and p^2 exceeds the length of the representative of p by $2(top - y)$, we can account for the cost by the length increase; note that the total length of the canonical representatives is always bounded by the size of the routing region.

At this point we have computed the bend sequences of p^1 and p^2. We then split the c-sequence of u and v at the appropriate places using the operation Access&Split. This takes amortized time $O(1)$ for each Access&Split and hence the cost is easily accounted for.

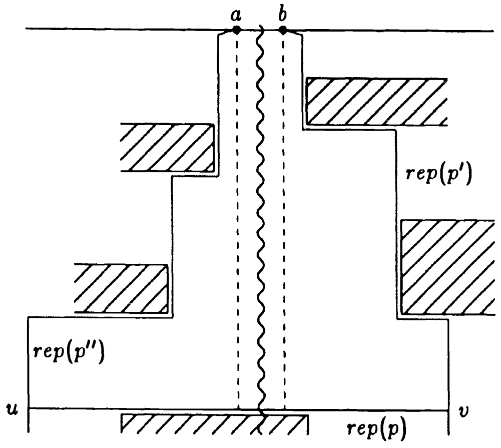


FIG. 19.

Finally, we observe that the ordering and consistency property is preserved.

Case 5.2 (no saturated cut found): Let S be any segment in FS. We route as shown in Fig. 20.

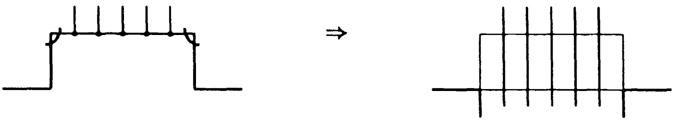


FIG. 20.

All cuts between s and t have their free capacity reduced by two and are added to a *satcut*-list if necessary. This completes the description of the various cases of the routing algorithm.

Suppose now that we processed row *top* completely. We unmark all cuts using the set of marked nets in time proportional to their number, decrease *top* by one, go through *Top* and adjust bend-sequences as described in case 1, and are ready for processing the new row *top*.

This completes the proof of the main theorem.

REFERENCES

[BM] M. BECKER AND K. MEHLHORN, *Algorithms for routing in planar graphs*, Acta Inform., 23 (1986), pp. 163–176.

[F] A. FRANK, *Disjoint paths in a rectilinear grid*, Combinatorica, 2 (1982), pp. 361–371.

[GT] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, in Proc. of the 15th SIGACT Symp., 1983, pp. 246–251.

[HMRT] K. HOFFMANN, K. MEHLHORN, P. ROSENSTIEHL, AND R. E. TARJAN, *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control, 68 (1986), pp. 170–184.

[IA] H. Imai and T. Asano, *Dynamic orthogonal segment intersection search*, J. Algorithms, 8 (1987), pp. 1–18.

[K] M. Kaufmann, *A linear-time algorithm for routing in a convex grid*, IEEE Trans. Computer-Aided Design, (1990), pp. 180–184.

[KM1] M. Kaufmann and K. Mehlhorn, *Routing through a generalized switchbox*, Journal Algorithms, 7 (1986), pp. 510–531.

[KM2] ———, *On local routing of two-terminal nets*, J. Combin. Theory Ser. B., (to appear).

[MP] K. Mehlhorn and F. P. Preparata, *Routing through a rectangle*, J. Assoc. Comput. Mach., 33 (1986), pp. 60–85.

[NSS] T. Nishizeki, N. Saito, and K. Suzuki, *A linear-time routing algorithm for convex grids*, IEEE Trans. Computer-Aided Design, CAD-4 (1985), pp. 68–76.

[Sh] A. Schrijver, *Edge-disjoint homotopic paths in straight-line planar graphs*, Universität des Saarlandes, preprint.

[Schw] Ch. Schwarz, *Ein Suchproblem in dynamischen Folgen*, Diplomarbeit, Universität des Saarlandes, 1989.

[W] B. Weinelt, *Homotopische Knock-Knee-Verdrahtung*: *Eine Linearzeit-Implementierung*, Diplomarbeit, Universität des Saarlandes, 1990.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 247–254, April 1994

SOME RESULTS ON ELUSIVE GRAPH PROPERTIES*

EBERHARD TRIESCH†

Abstract. This article proves several graph properties to be elusive. Two of the main results are

1. If $\mathcal{P}$ is a decreasing graph property containing no graph of girth smaller than 5, then $\mathcal{P}$ is elusive.
2. The property of having matching number at most k, $k < \lfloor |V|/2 \rfloor$, is elusive.

The proofs are all based on a topological method developed by Kahn, Saks, and Sturtevant.

Key words. monotone graph property, elusive, recognition complexity, girth, matching

AMS subject classifications. 68Q05, 68R05, 05C25

1. Introduction.

Let T be a finite set and $\mathcal{P}$ a property of subsets of T, i.e., $\mathcal{P} \subset 2^T$, the power set of T. Imagine two players $\mathcal{A}$ (*Algy*) and $\mathcal{S}$ (*Strategist*). Player $\mathcal{A}$ wants to learn from $\mathcal{S}$ whether an unknown set $X \subset T$ is in $\mathcal{P}$ or not by asking him questions of the form *Is* $x \in X$? ($x \in T$). The goal of $\mathcal{A}$ is to minimize the number of questions, but $\mathcal{S}$ wants to force $\mathcal{A}$ to ask as many questions as possible. If both players play optimally from their point of view, then the number of questions that are asked in the game is called the *recognition complexity* of $\mathcal{P}$ (also, *Boolean decision tree complexity*) and is denoted by $c(\mathcal{P})$. If $\mathcal{S}$ can force $\mathcal{A}$ to probe all elements of T, then $\mathcal{P}$ is called *elusive* (also, *evasive*).

The recognition complexity has been studied by various authors during the last 15 years (see all the references except [5], [10], [11], and [14]) and has turned out to be especially interesting for the case that T equals $V^{(2)}$, the set of two-element subsets of some finite set V. In this case the subsets of T can be interpreted as graphs with vertex set V. A *graph property* is a subset $\mathcal{P} \subset 2^T$, $T = V^{(2)}$, such that $\mathcal{P}$ contains with each graph G also each isomorphic copy of G (with vertex set V). A graph property $\mathcal{P}$ is called *nontrivial* if $\mathcal{P} \neq \emptyset$, $\mathcal{P} \neq 2^T$. It is called *decreasing* if, for each $G \in \mathcal{P}$, all subgraphs of G (with vertex set V) are contained in $\mathcal{P}$, *increasing* if $2^T \setminus \mathcal{P}$ is decreasing and *monotonic* if it is increasing or decreasing. In 1973, Rosenberg [12] conjectured that there exists some $\gamma > 0$ such that

$$c(\mathcal{P}) \geq \gamma n^2$$

for all nontrivial graph properties $\mathcal{P}$, $n = |V|$.

Meanwhile, there exist many counterexamples to the conjecture (see [2]). Together with Aanderaa, he modified his conjecture as follows.

There exists $\gamma > 0$ *such that for all nontrivial monotonic graph properties* $\mathcal{P}$ *we have* $c(\mathcal{P}) \geq \gamma n^2$.

This conjecture was proved by Rivest and Vuillemin [13] in 1975 with $\gamma = 1/16$. Kleitman and Kwiatkowski [8] improved the value of γ from 1/16 to 1/9 for n large. Then, in 1984, Kahn, Saks, and Sturtevant [9] proved that $c(\mathcal{P}) \geq \binom{n}{2} + o(n^2)$, which is strong support for the following conjecture of Karp that is still unresolved.

All nontrivial monotonic graph properties are elusive.

In fact, Kahn, Saks, and Sturtevant proved Karp's conjecture in case n is a prime power. Their method (which comes from algebraic topology) was applied by King [7] to digraph properties and by Yao [17] to monotone bipartite graph properties. We now use their method in a different way to obtain some new results, e.g., the theorem that all decreasing graph properties containing no graphs with cycles of length 3 or 4 are elusive.

*Received by the editors January 21, 1991; accepted for publication (in revised form) July 28, 1992.

†Lehrstuhl für Unternehmensforschung, RWTH Aachen, Templergraben 64, D-5100 Aachen, Germany.

2. Tools from algebraic topology. From now on, we assume that $\mathcal{P}$ is a monotone nontrivial graph property.

Because $c(\mathcal{P}) = c(2^T \backslash \mathcal{P})$, it suffices to study decreasing graph properties. But a decreasing graph property is an (abstract) simplicial complex. Hence, its *Euler characteristic* $\chi(\mathcal{P})$ is defined as

$$\chi(\mathcal{P}) := \sum_{i=1}^{|T|} (-1)^{i-1} a_i,$$

where $a_i := a_i(\mathcal{P}) := |\{X \in \mathcal{P} : |X| = i\}|$.

Now if G is an abelian group (here, $G = \mathbf{Z}$ or $G = \mathbf{Z}/p\mathbf{Z}$, p prime), the complex is called *G-acyclic* if

$$H_0(\mathcal{P}, G) = G \quad \text{and} \quad H_i(\mathcal{P}, G) = 0 \quad \text{for } i > 0,$$

where $H_i(\mathcal{P}, G)$ denotes the i-dimensional homology group of $\mathcal{P}$ with respect to G. Kahn, Saks, and Sturtevant established the following connection between complexity theory and algebraic topology.

THEOREM 1. *If $\mathcal{P}$ is not elusive, then it is $\mathbf{Z}_p$-acyclic for all primes p.*

Denote by $\mathcal{G}$ the set of all finite groups G containing a normal p-subgroup G_1, $G_1 \trianglelefteq G$, such that the factor group G/G_1 is cyclic. If G is a permutation group on T leaving $\mathcal{P}$ invariant, we define the simplicial complex $\mathcal{P}_G$ with vertex set $T_G := \{B \in \mathcal{P} : B \text{ orbit of } G\}$ by

$$\{B_1, \ldots, B_k\} \in \mathcal{P}_G :\Longleftrightarrow B_1 \cup \cdots \cup B_k \in \mathcal{P}.$$

Now the following theorem of Smith [14] and Olivier [11] is the essential tool in [9] and in what follows.

THEOREM 2. *If $G \in \mathcal{G}$ is a permutation group on T leaving the $\mathbf{Z}_p$-acyclic complex $\mathcal{P}$ invariant, then*

$$\chi(\mathcal{P}_G) = 1.$$

3. Some applications. We start by deriving some information about a cyclic group acting on $V^{(2)}$.

LEMMA 1. *Let $V := \mathbf{Z}_n := \mathbf{Z}/n\mathbf{Z}$ and $\tau : \mathbf{Z}_n \to \mathbf{Z}_n : x \to x + 1$. The group $G := \langle \tau \rangle$ generated by τ acts canonically on $V^{(2)}$ and the following statements about the orbits of G hold.*

(i) *If n is odd, $V^{(2)}$ decomposes into $(n-1)/2$ orbits of cardinality n.*

(ii) *If n is even, $V^{(2)}$ decomposes into $(n/2 - 1)$ orbits of cardinality n and one orbit of cardinality $n/2$.*

(iii) *The set of edges $\{\{0, j\} : 1 \le j \le \lfloor n/2 \rfloor\}$ is a system of distinct representatives for the orbits of G.*

(iv) *Denote by $D_j^n = D_j$ the orbit of $\{0, j\}$ and by $F_j^n = F_j$ the corresponding graph (V, D_j). Then F_j is the vertex disjoint union of s_j cycles of length r_j, where $s_j :=$ $\gcd(j, n)$ and $r_j := n/s_j$, $(1 \le j < n/2)$. For n even we have $F_{n/2} \simeq \frac{n}{2} K_2$. The components of F_j are the sets $M_{j,i}^n := M_{j,i} := \{x \in \mathbf{Z}_n : x \equiv i \bmod s_j\}, 0 \le i < s_j, 1 \le j \le \lfloor n/2 \rfloor$.*

Proof. We have $G = \langle \tau \rangle = \{\tau^j : 0 \le j < n\}$ and $\tau^j(x) = x + j, 0 \le j < n$. An edge $\{x, y\} \in V^{(2)}$ is a fixed point of τ^j if and only if $\{x + j, y + j\} = \{x, y\}$. This is certainly true for $j = 0$. For $0 < j < n$ we must have $x + j = y$ and $y + j = x$; hence, $x + y + 2j = x + y$ and therefore $2j = 0$ (in $\mathbf{Z}_n$), i.e., n is even and $j = n/2$. It is easy to see that there are exactly $n/2$ edges $\{x, y\}$ in $V^{(2)}$ such that $x + \frac{n}{2} = y$ and $y + \frac{n}{2} = x$.

From Burnside's Lemma we infer that the number of orbits is

$$\frac{1}{n}\sum_{0\leq j<n} (\#\text{ fixed points of } \tau^j \text{ in } V^{(2)}),$$

which equals $\frac{1}{n}\binom{n}{2} = \frac{n-1}{2}$ if n is odd and $\frac{1}{n}\left(\binom{n}{2} + \frac{n}{2}\right) = \frac{n}{2}$ if n is even.

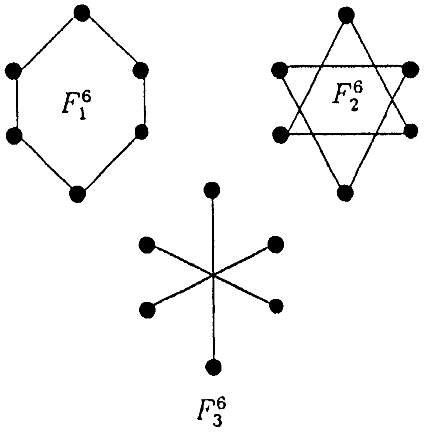


FIG. 1. *The orbits of Lemma* 1 *for* $n = 6$.

If $\{x, y\}$ and $\{x', y'\}$ are in the same orbit, then $x - y = \pm(x' - y')$. Hence, the edges $\{0, j\}$, $1 \leq j \leq \lfloor n/2 \rfloor$, belong to different orbits and (because we know the number of orbits) form a system of distinct representatives for the orbits and (iii) follows. To complete the proof of (i) and (ii), we note that $|D_j|$ is the index of the stabilizer of $\{0, j\}$ and hence (cf. the argument at the beginning of the proof)

$$|D_j| = \begin{cases} n & \text{for } j < n/2 \\ n/2 & \text{for } n \text{ even and } j = n/2. \end{cases}$$

Now G is a subgroup of the automorphism group $\operatorname{Aut}(F_j)$ of F_j and hence $\operatorname{Aut}(F_j)$ acts transitively on the vertices of F_j. We conclude that F_j is a regular graph of degree 2 for $j < n/2$ and of degree 1 for $j = n/2$. Hence, $F_{n/2} \simeq \frac{n}{2}K_2$ and F_j is a disjoint union of cycles for $j < n/2$. The cycle in F_j containing $\{0, j\}$ consists of the edges $\{0, j\}, \{j, 2j\}, \ldots, \{(r_j - 1)j, 0\}$ with

$$r_j = \min\{x \in \mathbf{Z} : x \geq 1, xj \equiv 0 \bmod n\}.$$

Again by the transitivity of $\operatorname{Aut}(F_j)$ on V, all cycles of F_j have the same length r_j and their number is $s_j = \gcd(j, n)$.

Finally, we note that x and y are joined by a path in F_j if and only if there exists some positive $k \in \mathbf{Z}$ such that $\tau^{kj}(x) = y$, i.e., if and only if there exists some $z \in \mathbf{Z}_n$ such that $x - y = zj$, which holds if and only if $x \equiv y \bmod \gcd(j, n)$. This proves (iv). □

Recall that the *girth* of a graph is the length of its shortest cycle (the girth of a forest being defined as ∞). We can now prove Theorem 3.

THEOREM 3. *Suppose $\mathcal{P}$ is a decreasing* (*nontrivial*) *graph property such that all graphs of $\mathcal{P}$ have girth greater than* 4. *Then $\mathcal{P}$ is elusive.*

Proof. If n is a power of 2, then elusiveness follows from the (more general) results in [9] (Karp's conjecture for prime powers). Hence, we may assume that some odd prime p divides n, $n = pm$ say, and $V = \mathbf{Z}_p \times \mathbf{Z}_m$. Let $G := \langle\tau\rangle$, the group generated by $\tau : \mathbf{Z}_p \to \mathbf{Z}_p : x \to x + 1$ and $H := \langle\sigma\rangle$, where $\sigma : \mathbf{Z}_m \to \mathbf{Z}_m : y \to y + 1$. Recall that the *wreath product* $G \wr H$ is a permutation group on V defined as follows:

$$G \wr H := \{(f; \pi) | f : \mathbf{Z}_m \to G, \pi \in H\}.$$

For $(i, j) \in V$ and $(f; \pi) \in G \wr H$, we have

$$(f; \pi)(i, j) := (f(\pi(j))(i), \pi(j)),$$

$G \wr H$ contains the normal subgroup

$$G_1 := \{(f; 1_H) | f : \mathbf{Z}_m \to G\},$$

which is isomorphic to the m-fold direct product $G^m = G \times \cdots \times G$. Hence, G_1 is a p-group and we have $(G \wr H)/G_1 \simeq H$. It follows that $G \wr H$ is in $\mathcal{G}$.

Now suppose there was a nonelusive graph property $\mathcal{P}$ that contained graphs of girth greater than 4 only. By Theorems 1 and 2 we must have

$$\chi(\mathcal{P}_{G\wr H}) = 1.$$

To obtain a contradiction, we consider two cases.

(a) $p = 3$. The orbit of an edge joining two vertices in some set $V_i := \mathbf{Z}_p \times \{i\}, i \in \mathbf{Z}_m$, is isomorphic to mK_3, the graph consisting of m disjoint triangles, by Lemma 1 and the action of the wreath product. But K_3 has girth 3. The orbit of an edge joining vertices in V_i and $V_j (i \neq j)$ is easily seen to contain the complete bipartite graph $K_{3,3}$ as a subgraph that has girth 4. We infer that $\mathcal{P}_G$ is empty, a contradiction.

(b) $p > 3$. Similarly to (a), the orbits of edges joining nodes in V_i and V_j, $i \neq j$, are easily seen to contain some complete bipartite graph $K_{p,p}$ that has girth 4 and therefore cannot be contained in $\mathcal{P}$. The orbits of edges joining two points in V_i are isomorphic to mC_p, the graph consisting of m disjoint copies of the p-cycle C_p, and there are $(p - 1)/2$ such orbits (cf. Lemma 1 and the action of the wreath product). We claim that the union of two such orbits contains a 4-cycle. It obviously suffices to show that each union $F_j \cup F_k$, $1 \leq j < k \leq (p - 1)/2$, contains a 4-cycle (notation of Lemma 1). But for each $x \in \mathbf{Z}_p$ the vertices $x, x + j, x + j + k, x + k$ lie on a 4-cycle in $F_j \cup F_k$ (in this order). It follows that $\chi(\mathcal{P}_{G\wr H}) = (p - 1)/2 > 1$, again a contradiction. □

COROLLARY 1. *Suppose $\mathcal{P}_k$ consists of all graphs having girth at least k. Then $\mathcal{P}_k$ is elusive for all $k \geq 4$.*

Proof. The case $k = 4$ (triangle-free graphs) was proved by Bollobás by exhibiting an explicit strategy for player $\mathcal{S}$ that forces $\mathcal{A}$ to probe all edges (cf. [3]). For $k > 4$ we apply Theorem 3. □

The next theorem is easy to prove.

THEOREM 4. *Suppose $\mathcal{P}$ is a decreasing property such that each graph in $\mathcal{P}$ has minimum degree at most one. Then $\mathcal{P}$ is elusive.*

Proof. Without loss of generality suppose that $n = |V|$ is not a power of two. Obviously, it suffices to exhibit a permutation group G on V such that each graph with nonempty edge set that is invariant under G has minimum degree at least two. For n odd, a transitive cyclic group will do by Lemma 1. For n even, n contains an odd prime divisor by assumption. Hence, we can choose a wreath product as in Theorem 3. □

We are now going to prove elusiveness for some graph properties that are often considered in graph theory.

THEOREM 5. *Let $\mathcal{P}_k$ denote the set of graphs with vertex set V and maximum degree at most k. Then $\mathcal{P}_k$ is elusive for $0 \le k \le n-2$.*

Proof. Suppose first that n is odd. We choose a cyclic and transitive permutation group on V (without loss of generality $V = \mathbf{Z}_n$ and $G = \langle \tau \rangle$ as in Lemma 1). Obviously, $G \in \mathcal{G}$. Now suppose that $\mathcal{P}_k$ is not elusive. By Theorem 2, $\chi((\mathcal{P}_k)_G) = 1$. A union $\bigcup_{i=1}^{m} F_{j_i}$ of distinct orbits is in $\mathcal{P}_k$ if and only if $m \le \lfloor k/2 \rfloor =: u$. Hence, letting $t := (n-1)/2$, we have

$$1 - \chi((\mathcal{P}_k)_G) = \sum_{i=0}^{u} (-1)^i \binom{t}{i}$$

$$= \sum_{i=0}^{u} (-1)^i \left[\binom{t-1}{i-1} + \binom{t-1}{i} \right]$$

$$= (-1)^u \binom{t-1}{u} \neq 0$$

because $t - 1 = (n-3)/2 \ge \lfloor k/2 \rfloor$ for n odd, $k \le n-2$, a contradiction.

Now assume that n is even. If k is even we can proceed completely analogous to the preceding case choosing $V := \mathbf{Z}_n$, $G := \langle \tau \rangle$ as in Lemma 1. Again, a union of distinct orbits is in $\mathcal{P}_k$ if and only if $m \le k/2$. Letting $t := n/2$, $u := k/2$ we get

$$1 - \chi((\mathcal{P}_k)_G) = (-1)^u \binom{t-1}{u} \neq 0.$$

If k is odd, which implies $k \le n-3$, we let $V := \{a\} \dot{\cup} \mathbf{Z}_{n-1}$ and $G := \langle \tau \rangle$, where $\tau(a) := a$ and $\tau(x) := x + 1$ for $x \in \mathbf{Z}_{n-1}$. The orbits of G on $V^{(2)}$ are the sets D_j^{n-1}, $1 \le j \le \frac{n}{2} - 1$ and $D_0 := \{\{a, j\} : j \in \mathbf{Z}_{n-1}\}$. Because $k \le n-3$, no graph in $\mathcal{P}_k$ contains all the edges in D_0. Now elusiveness follows just as in the first part of the proof. □

Denoting by $\Delta(H)$ and $\delta(H)$ the maximal and minimal degree of a graph H, respectively, we obtain the following.

COROLLARY 2. *The graph properties $\{H : \Delta(H) \ge k\}$ and $\{H : \delta(H) \ge k\}$ are elusive for $1 \le k \le n-1$; $\{H : \delta(H) \le k\}$ is elusive for $0 \le k \le n-2$.*

Proof. If $\mathcal{P}$ is a graph property, denote by $\mathcal{P}^*$ the set $\{H = (V, E) : \bar{H} := (V, V^{(2)} \setminus E) \in \mathcal{P}\}$. It is well known that $c(P) = c(P^*) = c(2^T \setminus \mathcal{P})$ (cf. [15]). Hence, the result follows from Theorem 5. □

Suppose $H = (V, E)$ is a graph. A *matching* of H is a set M of edges ($M \subset E$) such that no two of them have an endpoint in common. A vertex x is covered by M if x is an endpoint of one of the edges in M. A matching covering all the nodes in V is called a *perfect matching*. (Equivalently, we say that H has a *1-factor*.) H is called *factor-critical* if $H - x$, the graph obtained from H by deleting the vertex x, has a *1-factor* for all $x \in V$. The *matching number* $\nu(H)$ of H is the maximum cardinality of a matching in H. The following lemma is well known (cf. [10, Ex. 3.2.5 and Thm. 5.5.24]).

LEMMA 2. *Suppose H is a graph whose automorphism group is transitive on the vertices. Then each component of H is either factor-critical or contains a 1-factor.*

As an application, we get the following.

LEMMA 3. *Let n, p, m, G, H, and $V = \mathbf{Z}_p \times \mathbf{Z}_m$ be defined as in the proof of Theorem* 1 *except that the prime $p = 2$ is allowed. Then the following results hold for an arbitrary orbit B of $G \wr H$ acting on $V^{(2)}$:*

(i) *In case $p = 2$, the graph (V, B) contains a* 1*-factor.*

(ii) *In case $p > 2$, the inequality $2\nu((V, B)) \geq n - m$ holds.*

Proof. Note that the orbit B of an edge e in $V^{(2)}$ is obtained as follows: if e joins two vertices in some set $V_i := \mathbf{Z}_p \times \{i\}$, then (V, B) is isomorphic to mK_2 in case $p = 2$ and to mC_p for $p > 2$. Hence, the result is clear in this case.

If e joins vertices in different V_i's, in V_0 and V_j say, consider the orbit D_j^m (and the corresponding graph F_j^m) of H acting on $\mathbf{Z}_m$. Then (V, B) arises from F_j^m by substituting the set V_i for each $i \in \mathbf{Z}_m$ and joining vertices in V_i and V_k if and only if $i \neq k$ and i and k are joined in F_j^m (see Figs. 1 and 2). It is clear that (V, B) satisfies the hypothesis of Lemma 2. Hence, for $p = 2$, each component has an even number of vertices and thus contains a perfect matching. For $p > 2$, just note that the number of components of (V, B) is $\gcd(j, m) \leq m/2$ and that each component either has a 1-factor or is factor-critical. □

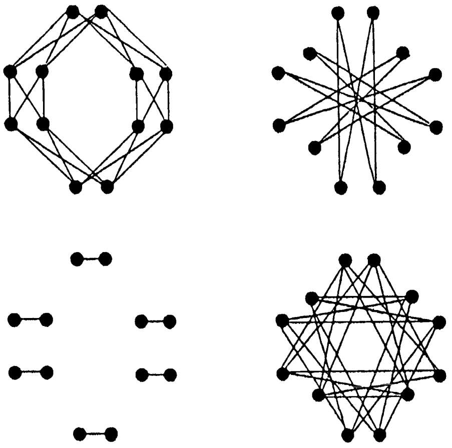


FIG. 2. *The orbits of $G \wr H$ on* $(\mathbf{Z}_2 \times \mathbf{Z}_6)^{(2)}$.

THEOREM 6. *Suppose k is a natural number satisfying $k < \lfloor n/2 \rfloor$ and denote by $\mathcal{P}_k$ the property*

$$\mathcal{P}_k := \{H : H \text{ is a graph with vertex set } V \text{ and } \nu(H) \leq k\}.$$

Then $\mathcal{P}_k$ is elusive.

Proof. If n is even, $n = 2m$ say, we choose a wreath product $G \wr H$ on $V := \mathbf{Z}_2 \times \mathbf{Z}_m$ as in Lemma 3. By Lemma 3(i), no orbit graph (V, B) is in $\mathcal{P}_k$ and thus $\chi((\mathcal{P}_k)_{G \wr H}) = 0$, which implies that $\mathcal{P}_k$ is elusive.

Now assume that n is odd, n no prime power. Hence, $n \geq 15$ and n has a prime divisor $q \geq 5$. By choosing again a wreath product $G \wr H$ of transitive cyclic groups on $\mathbf{Z}_q \times \mathbf{Z}_{n/q}$, we see that $\chi((\mathcal{P}_k)_{G \wr H}) = 0$ if $2k \leq n - \frac{n}{q} = n(1 - \frac{1}{q})$. So suppose that $2k > n(1 - \frac{1}{q}) \geq \frac{4}{5}n$. By Bertrand's postulate (cf. [5, p. 71]), we can choose a prime number p satisfying $n/3 < p \leq 2n/3$. (Note that p does not divide n.) We write $V = U \dot{\cup} W$ with $|U| = p$ and $|W| = n - p$. Because $|W|$ is even, we identify W with $\mathbf{Z}_2 \times \mathbf{Z}_{(n-p)/2}$ and choose a wreath product of cyclic groups as above on W and call that group G'. On U, we choose a cyclic transitive permutation group G'' and let $G := G' \times G''$. Recall that G' has a normal subgroup G_1' of order 2^r, $r := (n-p)/2$, and $G'/G_1' \simeq \mathbf{Z}_r$. Hence, G_1' is normal in G and $G/G_1' \simeq (G'/G_1') \times G'' \simeq \mathbf{Z}_r \times \mathbf{Z}_p \simeq \mathbf{Z}_{rp}$ because p and r are relatively prime. We conclude that $G \in \mathcal{G}$.

Now, the action of G on $V^{(2)}$ yields the following orbits:

(a) The orbits of G' on $W^{(2)}$, $W_1, \ldots, W_s$. By Lemma 3(i), each graph (W, W_i) has a 1-factor ($s = \lfloor r/2 \rfloor + 1$).

(b) The orbits $U_1, \ldots, U_t$ of G'' on $U^{(2)}$, where each graph (U, U_j) is a p-cycle ($t = (p-1)/2$).

(c) The orbit $B := \{\{u, w\} : u \in U, w \in W\}$.

Because $k < \lfloor n/2 \rfloor$, no graph $(V, W_i \cup U_j)$ is in $\mathcal{P}_k$. From $2k > \frac{4}{5}n > \frac{2}{3}n \geq \max(p, n-p)$, we infer that each of the graphs (V, W_i), (V, U_j) is in $\mathcal{P}_k$, $(1 \leq i \leq s, 1 \leq j \leq t)$. For (V, B), we have the following two cases.

(i) $k < \min(p, n-p)$. Then $(V, B) \notin \mathcal{P}_k$, implying

$$\chi((\mathcal{P}_k)_G) = \sum_{i=1}^{s}(-1)^{i-1}\binom{s}{i} + \sum_{j=1}^{t}(-1)^{j-1}\binom{t}{j} = 1 + 1 = 2,$$

and hence $\mathcal{P}_k$ is elusive.

(ii) $k \geq \min(p, n-p)$. Suppose first that $p = \min(p, n-p)$. Because $2k < n$ and $p + (n-p) = n$, we must have $k < n - p$. Thus, all graphs $(V, B \cup C)$, where C is a union of some of the orbits $U_1, \ldots, U_t$, are in $\mathcal{P}_k$. On the other hand, all graphs $(V, B \cup W_i)$, $(1 \leq i \leq s)$ have matching number $(n-1)/2$ and therefore are not in $\mathcal{P}_k$. We conclude that

$$\chi((\mathcal{P}_k)_G) = \sum_{i=1}^{s}(-1)^{i-1}\binom{s}{i} + \sum_{j=1}^{t}(-1)^{j-1}\binom{t}{j} + \sum_{j=0}^{t}(-1)^{j}\binom{t}{j} = 1 + 1 + 0 = 2.$$

Hence, $\mathcal{P}_k$ is not elusive. The case $n - p = \min(p, n-p)$ is similar.

This completes the proof. □

Again, the corresponding result for the properties $\{H : \nu(H) \geq k\}$ follows by taking complements.

The proof of our final result uses the same construction as the proof of Theorem 5 and is therefore left to the reader.

THEOREM 7. *For k even, $k < n$, denote by $\mathcal{P}_k$ the set of all graphs on V having a k-factor, i.e., a k-regular spanning subgraph. Then $\mathcal{P}_k$ is elusive.*

We conclude by mentioning that there are several important monotone graph properties that have not been shown to be elusive to the knowledge of the author. Two examples are *Hamiltonicity* and *k-connectedness* for $k \geq 3$.

Acknowledgments. I thank W. Obershelp for introducing me to the topic and for some very helpful discussions. I also thank the referee who helped to improve the article by several constructive suggestions.

REFERENCES

[1] M. Aigner, *Combinatorial Search*, Wiley-Teubner, Stuttgart and Chichester, 1988.

[2] M. R. Best, P. van Emde Boas, and H. W. Lenstra, Jr., *A sharpened version of the Aanderaa–Rosenberg conjecture*, Math. Centrum Tracts, Amsterdam, 1974.

[3] B. Bollobás, *Complete subgraphs are elusive*, J. Combin. Theory Ser. B, 21 (1976), pp. 1–7.

[4] ———, *Extremal Graph Theory*, Academic Press, London, 1978.

[5] K. Chandrasekharan, *Introduction to Analytic Number Theory*, Springer, Berlin, 1968.

[6] D. Grieser, *Some results on the complexity of sets*, Discrete Math., 88 (1991), pp. 179–192.

[7] V. King, *A lower bound for the recognition of digraph properties*, Combinatorica, 10 (1990), pp. 53–59.

[8] D. J. Kleitman and D. J. Kwiatkowski, *Further results on the Aanderaa–Rosenberg conjecture*, J. Combin. Theory Ser. B, 28 (1980), pp. 85–95.

[9] J. Kahn, M. Saks, and D. Sturtevant, *A topological approach to evasiveness*, Combinatorica, 4 (1984), pp. 297-306.

[10] L. Lovász and M. D. Plummer, *Matching Theory*, North Holland, Amsterdam, 1986.

[11] R. Olivier, *Fixed point sets of group actions on finite acyclic complexes*, Comment. Math. Helv., 50 (1975), pp. 155–177.

[12] A. L. Rosenberg, *On the time required to recognize properties of graphs: a problem*, SIGACT News, 5 (1973), pp. 15–16.

[13] R. L. Rivest and J. Vuillemin, *On recognizing graph properties from adjacency matrices*, Theoret. Comput. Sci., 3 (1976/77), pp. 371–384.

[14] P. A. Smith, *Fixed point theorems for periodic transformations*, Amer. J. Math., 63 (1941), pp. 1–8.

[15] E. Triesch, *Elusive properties*, in Combinatorial Theory, Proceedings SchloßRauischholzhausen, D. Jungnickel and K. Vedder, eds., Springer Lecture Notes in Math, 1982, pp. 321–326.

[16] ———, *Über die Komplexität von Grapheneigenschaften*, Dissertation, Aachen, 1984.

[17] A. C-C. Yao, *Monotone bipartite graph properties are evasive*, SIAM J. Comput., 17 (1988), pp. 517–520.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 255–260, April 1994

CLOSENESS OF NP-HARD SETS TO OTHER COMPLEXITY CLASSES*

BIN FU[†] AND HONG-ZHOU LI[‡]

Abstract. Let A be a language and C be a class of languages. A is said to be *s-C-close* (*s-C-outside-close*) if there exists $B \in C$ such that $\| (A \bigtriangleup B)^{\leq n} \| \leq s(n)$ (and $A \subseteq B$). If A is *q-C-close* (*q-C-outside-close*) for some polynomial q then it is simply said that A is *C-close* (*C-outside-close*). The following results are shown in this paper.

(1) No *NP-hard* set can be *coNP-close* unless $NP = coNP$.

(2) No *NP-hard* set can be *R-close* unless $NP = R$.

(3) No *NP-hard* set can be $O(\log n)$*-UP-close* unless $NP = FewP$.

(4) No *NP-hard* set can be $O(\log n)$-$C_=P$*-close* unless $NP \subseteq C_=P$.

(5) No *NP-hard* set can be *UP-outside-close* unless $NP = FewP$.

(6) No *NP-hard* set can be $C_=P$*-outside-close* unless $NP \subseteq C_=P$.

Key words. symmetric difference, sparse sets, *NP-hard* sets

AMS subject classification. 68C25

1. Introduction.

Research about the connection and difference between complexity classes are two main research lines in complexity theory. In recent years investigations about many counting classes have revealed surprising connections between complexity classes. In this paper we pay attention to the difference between *NP* and other complexity classes. We study whether an *NP-hard* set can be approximated sufficiently by the sets in other complexity classes.

Yesha [17] first considered measuring the similarity of two sets A and B by the density of their symmetric difference $A \bigtriangleup B$. Yesha [17] and Schöning [10] defined two sets A and B to be *s-close* to each other if for each n, $\| (A \bigtriangleup B)^{\leq n} \| \leq s(n)$. Can a *NP-hard* set be polynomially close to some set in P? Yesha and Schöning provided only partial answers to this question. Yesha proved that no *NP-hard* sets can be $O(\log \log n)$*-close* to any set in P unless $P = NP$. Schöning showed that no paddable *NP-hard* sets can be polynomially close to any set in P unless $P = NP$. Watanabe [16] showed that if $NP \subseteq P_{1-tt}(sparse)$ then $NP = R$. The question was not settled until Ogiwara and Watanabe [9] showed that if $NP \subseteq P_{btt}(sparse)$ then $P = NP$; the result follows since any set that is close to a set in P is $1 - tt$ reducible to a sparse set. Recently Fu [5] investigated lower bounds of closeness between many complexity classes. He showed that if an *NP-hard* set is the union of a set in $P_{btt}(sparse)$ and set A, then $NP \subseteq P_{dtt}(A)$. Thus no *NP-hard* set can be the union of a sparse set and a set in *coNP* (*FewP*) unless $NP = coNP$ ($NP = FewP$).

In this paper, we investigate the closeness between *NP-hard* sets and the sets of some other complexity classes such as *UP*, *coNP*, $C_=P$, and R. In order to characterize the conditions necessary for an *NP-hard* set to have a small density symmetric difference with a set A, we introduce two reductions: $\leq^P_{d\text{-maj}}$ and $\leq^P_{d\text{-conj}}$, both of which are positive truth-table reductions [11]. We show that if an *NP-hard* set can be polynomially close ($O(\log n)$*-close*) to set A then $NP \subseteq P_{d\text{-maj}}(A)$ (respectively, $NP \subseteq P_{d\text{-conj}}(A)$), and if *NP-hard* set H is a subset of A and A-H is sparse then $NP \subseteq P_{d\text{-conj}}(A)$.

*Received by the editors February 4, 1992; accepted for publication (in revised form) August 10, 1992.

[†] Beijing Computer Institute and Beijing Laboratory of Cognitive Science, Beijing, 100044, People's Republic of China. This work was supported in part by HTP863. Reference [6] is the earlier version of this paper.

[‡] Department of Mathematics, Yunnan Education College, Kunming, 650031, People's Republic of China. This research was performed while the author was visiting Beijing Computer Institute.

2. Preliminaries. We fix $\Sigma = \{0, 1\}$ as our alphabet . By "string" we mean an element of Σ^*. For a string x in Σ^*, $\mid x \mid$ denotes the length of x. We consider a standard canonical order on Σ^*. For any strings x and y, x is canonically smaller than y (write $x \leq y$) if either (1) $\mid x \mid < \mid y \mid$, or (2) $\mid x \mid = \mid y \mid$ and if there exists some k, $1 \leq k \leq \mid x \mid$, such that (for all $i : 1 \leq i < k[x_i = y_i]$ and $x_k = 0 \wedge y_k = 1$), where x_i is the ith symbol of the string x. For an $S \subseteq \Sigma^*$, the cardinality of S is denoted by $\parallel S \parallel$; set $S^{=n}(S^{\leq n})$ consists of all words of length $= n(\leq n)$ in S. In particular, we let $\Sigma^n = \{x : x \in \Sigma^*$ and $\mid x \mid = n\}$ and $\Sigma^{\leq n} = \{x : x \in \Sigma^*$ and $\mid x \mid \leq n\}$. Let $x, y \in \Sigma^*$; we define the interval $[x, y] : [x, y] = \{z : x \leq z \leq y$ and $z \in \Sigma^*\}$.

Let $[a, b]$ and $[c, d]$ be two disjoint intervals; we say $[a, b] \leq [c, d]$ if $b \leq c$. We use λ to denote the null string. N represents the set $\{0, 1, 2, \ldots\}$.

We use the pairing function $\langle . , . \rangle : \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. It is convenient to assume that for any x, y in Σ^*, $\mid < x, y > \mid \leq 2(\mid x \mid + \mid y \mid)$.

Our computation model is the Turing machine. Now we involve the following complexity classes: P (respectively, NP) denotes the class of languages accepted by deterministic (respectively, nondeterministic) Turing machines in polynomial-time. PF denotes the class of polynomial-time computable functions. $coNP = \{\Sigma^* - A : A \in NP\}$. UP [14] denotes the class of languages accepted by polynomial-time nondeterministic Turing machines that have at most one accepting path on each input. $FewP$ [1] represents the class of languages accepted by polynomial-time nondeterministic Turning machines that have at most polynomial number of accepting paths on each input. $C_=P$ was defined by Wagner [15]. A language L belongs to $C_=P$ if and only if there exists a polynomial-time nondeterministic Turing machine M such that $x \in L$ if and only if the number of accepting paths is equal to the number of rejecting paths. R (see [2]) represents one-sided bounded error probability complexity classes.

It is well known that $P \subseteq UP \subseteq FewP \subseteq NP$ and $P \subseteq R \subseteq NP$. In [13] relativizations are presented under which the classes NP and $C_=P$ are incomparable.

We now define some notions of polynomial-time reducibilities.

A $\leq_m^P$-reduction of A to B is a polynomial-time computable function f such that for each $x \in \Sigma^*, x \in A \Longleftrightarrow f(x) \in B$.

A $\leq_{k-tt}^P$-reduction of A to B is a pair $\langle f, g \rangle$ of polynomial-time computable functions such that the following hold for each $x \in \Sigma^*$: (1) $f(x) = \langle x_1, \ldots, x_k \rangle$ is an ordered k-*tuple* of strings, (2) $g(x)$ is a k-argument truth-table: $\{0, 1\}^k \longrightarrow \{0, 1\}$, (3) $x \in A \Longleftrightarrow g(x)(\chi_B(x_1), \ldots, \chi_B(x_k)) = 1$,where χ_B is the characteristic function of the set B.

A $\leq_{btt}^P$-reduction of A to B is a $\leq_{k-tt}^P$-reduction of A to B for some integer k.

A $\leq_{dtt}^P$-reduction of A to B is a polynomial-time computable function f such that for each $x \in \Sigma^*$, $f(x)$ is a list of strings in Σ^* and $x \in A \Longleftrightarrow$ one of the strings of $f(x)$ is in B.

A $\leq_{d-\mathrm{maj}}^P$-reduction of A to B is a polynomial-time computable function f such that for each $x \in \Sigma^*$, $f(x) = \langle C_1, \ldots, C_k \rangle$, where C_i is a list of strings in Σ^* and $x \in A \Longleftrightarrow \parallel B \bigcap C_i \parallel > \parallel C_i \parallel /2$ for some $i \leq k$.

A $\leq_{d-\mathrm{conj}}^P$-reduction of A to B is a polynomial-time computable function f such that for each $x \in \Sigma^*$, $f(x) = \langle C_1, \ldots, C_k \rangle$, where C_i is a list of strings in Σ^* and $x \in A \Longleftrightarrow C_i \subseteq B$ for some $i \leq k$.

A $\leq_{tt}^P$-reduction of A to B is a pair of polynomial-time computable functions $\langle f, g \rangle$ such that for each $x \in \Sigma^*$, $f(x) = \langle x_1, \ldots, x_k \rangle$, $g(x)$ is a polynomial-time boolean circuit with k inputs, and $x \in A \Longleftrightarrow g(x)(\ \chi_B(x_1), \ldots, \chi_B(x_k)) = 1$.

For a boolean circuit $C : \{0, 1\}^k \longrightarrow \{0, 1\}$, if for any two k-tuples of $\{0, 1\} : \langle a_1, \ldots, a_k \rangle$ and $\langle b_1, \ldots, b_k \rangle$, with for all $i \leq k\ (a_i \leq b_i)$, $C(a_1, \ldots, a_k) = 1$ implies $C(b_1, \ldots, b_k) = 1$, then we say C is positive.

If $A \leq_{tt}^P B$ via $\langle f, g \rangle$ and for each $x \in \Sigma^*$ $g(x)$ is positive, then we say $A \leq_{ptt}^P B$. The reduction $\leq_{ptt}^P$ is called a positive truth-table reduction.

Let C be a class of languages, and let $\leq_r^P$ be a type of reduction (e.g., $r = m, btt, k-tt, ptt$, etc.). Then for any $A \subseteq \Sigma^*$, we define $A \in P_r(C)$ if and only if $A \leq_r^P B$ for some B in C. If every language in C is *$\leq_r^P$-reducible* to H, then we say H is *C-r-hard*.

For any set $A \subseteq \Sigma^*$, we say A is sparse if there exists a polynomial p such that $\| A^{\leq n} \| \leq p(n)$ for every $n \in N$. Let languages $A, B \subseteq \Sigma^*$; we define $A \triangle B = (A - B) \cup (B - A)$. The function $\mathrm{dist}_{A,B} : N \longrightarrow N$ is called the distance function of A and B, where $\mathrm{dist}_{A,B}(n) = \| (A \triangle B)^{\leq n} \|$. Let C be a class of languages, and let function $s : N \longrightarrow N$; A is said to be *s-C-close* (*s-C-outside-close*, *s-C-inside-close*) if there exists $B \in C$ such that $\mathrm{dist}_{A,B}(n) \leq s(n)$ (and $A \subseteq B$, $B \subseteq A$ respectively). If A is *q-C-close* (*q-C-outside-close*, *q-C-inside-close*) for some polynomial q then we simply say A is *C-close* (*C-outside-close*, *C-inside-close*). We assume all polynomials involved in this paper are monotonic.

3. Closeness to *NP-m-hard* sets. In this section we study whether an *NP-hard* set can be *$s(n)$-close* to some other complexity classes for some slowly increasing function $s(n)$. First we develop a useful technical lemma that generalizes many results.

LEMMA 3.1. *Let H be NP-m-hard, $K \in NP$, $A \subseteq \Sigma^*$, and $\mathrm{dist}_{A,H}(n) \leq s(n)$, where $s(n)$ is a nondecreasing function in PF. There exists a polynomial $p_0(n)$ such that for every polynomial-time computable function $h(n) > 2s(p_0(n))$ there is an algorithm that has the following properties for input x of length n:*

(1) *The algorithm either accepts x or outputs a series of sets: $G_1, \ldots, G_{e_x}$, where $\| G_e \| \leq h(n)$ and $G_e \subseteq \Sigma^{\leq p_0(n)}$ for all $e \leq e_x$.*

(2) *$x \in K \Longleftrightarrow$ (the algorithm accepts x) or ($\| A \bigcap G_e \| > s(p_0(n))$ for some $e \leq e_x$).*

(3) *If $\| \overline{A} \bigcap G_e \| > s(p_0(n))$ for all e with $\| G_e \| = h(n)$, then $x \in K \Longleftrightarrow$ the algorithm accepts x.*

(4) *The algorithm will stop in $p_1(h(n)) + p_1(n)$ steps, where $p_1(n)$ is a polynomial.*

Proof. Since $K \in NP$, there exist a set $C \in P$ and a polynomial $r(n)$ such that $x \in K \Longleftrightarrow$ there exists $w (w \in \Sigma^{\leq r(|x|)} \wedge \langle x, w \rangle \in C)$. We define the left set $L(C, r)$ as in [9]:

$L(C, r) = \{\langle x, y \rangle : (y \in \Sigma^{\leq r(|x|)}) \wedge (\text{there exists } w \in \Sigma^{\leq r(|x|)} (y \leq w)) \wedge (\langle x, w \rangle \in C)\}$.

It is easy to see $L(C, r) \in NP$. Because H is *NP-m-hard*, there exists a function $f \in PF$ such that $L(C, r) \leq_m^P H$ via f. Let f be computable in polynomial-time $t(n)$. For a fixed $x \in \Sigma^*$, if there exists a $w \in \Sigma^{\leq r(|x|)}$ such that $\langle x, w \rangle \in C$, then we let $w_{\max}$ be the largest such string. □

Let $p_0(n) = t(2(n + r(n)))$.

ALGORITHM.

Input x of length n.

Let $U_0 := \{[\lambda, 1^{r(n)}]\}$. $e := 0$.

Repeat

Let $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$ be all the intervals in U_e.

It will follow that these intervals are all disjoint.

U_e is partitioned into the blocks $U_{e,1}, U_{e,2}, \ldots$, which satisfy the following three conditions.

(1) $U_e = U_{e,1} \cup U_{e,2} \cup \cdots \cup U_{e,m_e}$, where m_e is the number of blocks.

(2) Any two intervals $[a, b]$, $[a', b']$ in U_e are in the same block if and only if $f(\langle x, a \rangle) = f(\langle x, a' \rangle)$.

(3) For any two blocks $U_{e,i}, U_{e,j}$, $i < j$, $[a_i^{(0)}, b_i^{(0)}] < [a_j^{(0)}, b_j^{(0)}]$, where $[a_i^{(0)}, b_i^{(0)}]$ and $[a_j^{(0)}, b_j^{(0)}]$ are the least intervals (leftmost intervals) of $U_{e,i}$ and $U_{e,j}$ respectively.

Let $[a_i^{(1)}, b_i^{(1)}]$ be the largest interval (rightmost interval) in $U_{e,i}$ ($i \leq m_e$).

$V_e := \{[a_i^{(1)}, b_i^{(1)}] : i \leq h(n)\}$

$G_e := \{f(\langle x, a_i^{(1)} \rangle) : a_i^{(1)}$ is the left point of one of the intervals in $V_e\}$

```
    $U_{e+i} := \bigcup_{t\in V_e}\{t_1, t_2 : t_1, t_2$ are the first and last half of $t$ respectively$\}$
    $e := e + 1$
Until all the intervals in $U_e$ are of width 1
$e_x := e - 1$ {For $e$ was added an extra one before exiting the cycle}
If There is $b \in \bigcup_{t\in U_{e_x}}$ such that $\langle x, b\rangle \in C$.
Then Accept $x$.
Else Output: $G_1, \ldots, G_{e_x}$.
End of the Algorithm.
```

Claim 1. (i) For all $e \le e_x$, there are at most $2h(n)$ intervals in U_e. (ii) The cycle of the algorithm will not be repeated more than $r(n)+1$ times. (iii) $G_e \subseteq \Sigma^{\le p_0(n)}$ and $\| G_e \| \le h(n)$. (iv) $\| G_e \| = \| V_e \|$.

Proof of Claim 1. From the algorithm (i), (ii), and (iv) hold clearly. We must only prove (iii). For each interval $[c_i, d_i]$ in V_e, $[c_i, d_i] \subseteq [\lambda, 1^{r(n)}]$. Hence $| c_i | \le r(n)$ and $| \langle x, c_i\rangle | \le 2(n+ | c_i |) \le 2(n + r(n))$. Because f is computable in time t, $| f(\langle x, c_i\rangle) | \le t(2(n + r(n))) = p_0(n)$. So $G_e \subseteq \Sigma^{\le p_0(n)}$. □

Claim 2. If for some $e \le e_x$, $\| G_e \bigcap A \| > s(p_0(n))$, then $x \in K$.

Proof of Claim 2. Clearly for any two different intervals $[c_i, d_i]$, $[c_j, d_j]$ in V_e, they belong to different blocks $U_{e,i}$, $U_{e,j}$ respectively. Thus $f(\langle x, c_i\rangle) \ne f(\langle x, c_j\rangle)$. By Claim 1 (iii) $G_e \subseteq \Sigma^{\le p_0(n)}$. Since $\| (A \bigtriangleup H)^{\le p_0(n)} \| \le s(p_0(n))$ and $\| G_e \bigcap A \| > s(p_0(n))$, $H \bigcap G_e \ne \emptyset$, and there are some $c_i \in \Sigma^{\le r(n)}$ with $f(\langle x, c_i\rangle) \in H$. So $x \in K$. □

Claim 3. Let $e \le e_x$. If $\| G_e \| < h(n)$, then $w_{\max}$ is in an interval of $U_e \Longleftrightarrow w_{\max}$ is in an interval of V_e.

Proof of Claim 3. We assume that $w_{\max}$ is in one of the intervals of the block U_{e,i_0} ($i_0 \le m_e$). For any two intervals $[c, d]$, $[c', d']$ in the same block, $f(\langle x, c\rangle) = f(\langle x, c'\rangle)$. So $w_{\max}$ must be in $[a_{i_0}^{(1)}, b_{i_0}^{(1)}]$, which is the largest interval in U_{e,i_0}.

By Claim 1 (iv) $\| V_e \| = \| G_e \| < h(n)$; thus $\| V_e \| = m_e$. For every block $U_{e,i}$ ($i \le m_e$), its largest interval $[a_i^{(1)}, b_i^{(1)}]$ belongs to V_e. Hence $w_{\max}$ belongs to an interval in V_e. □

Claim 4. Let $e \le e_x$. If $\| G_e \bigcap \overline{A} \| > s(p_0(n))$, then $w_{\max}$ is in an interval of $U_e \Longleftrightarrow$ $w_{\max}$ is in an interval of V_e.

Proof of Claim 4. If $\| G_e \| < h(n)$, this claim holds by Claim 3. We only discuss the case $\| G_e \| = h(n)$. By Claim 1 (iv), $\| G_e \| = \| V_e \|$. We assume that $w_{\max}$ is in one of the intervals of the block U_{e,i_0} ($i_0 \le m_e$). By the same reasons as Claim 3 $w_{\max}$ must be in $[a_{i_0}^{(1)}, b_{i_0}^{(1)}]$. By Claim 1 (iii), $G_e \subseteq \Sigma^{\le p_0(n)}$. Since $\| (A \bigtriangleup H)^{\le p_0(n)} \| \le s(p_0(n))$ and $\| G_e \cap \overline{A} \| > s(p_0(n))$, we can conclude that $\overline{H} \bigcap G_e \ne \emptyset$, and there are some intervals $[c_{i_1}, d_{i_1}]$ in V_e such that $f(\langle x, c_{i_1}\rangle) \notin H$. Thus we have $i_1 \le h(n)$ with $f(\langle x, a_{i_1}^{(0)}\rangle) \notin H$ and $w_{\max} < a_{i_1}^{(0)}$. So $i_0 < i_1 \le h(n)$ and $w_{\max}$ belongs to an interval in V_e. □

Claim 5. If $\| G_e \bigcap \overline{A} \| > s(p_0(n))$ for all $e \le e_x$ with $\| G_e \| = h(n)$, then $x \in K \Longleftrightarrow$ the algorithm accepts x.

Proof of Claim 5. By the algorithm and Claim 4. □

Claim 6. $x \in K \Longleftrightarrow$ (The algorithm accepts x) or ($\| A \bigcap G_e \| > s(p_0(n))$ for some $e \le e_x$).

Proof of Claim 6. $\Longleftarrow$ By algorithm and Claim 2 it is easy to see.

$\Longrightarrow$ Let $x \in K$. We assume that the algorithm does not accept x.

It is impossible that $\| G_e \| < h(n)$ for all $e \le e_x$, otherwise the algorithm will accept x by Claim 3.

It is also impossible that $\| \overline{A} \bigcap G_e \| > s(p_0(n))$ for all $e \le e_x$ with $\| G_e \| = h(n)$, otherwise the algorithm will accept x by Claim 5.

Therefore, there exists an $e \le e_x$ such that $\| \overline{A} \bigcap G_e \| \le s(p_0(n))$ and $\| G_e \| = h(n)$. Hence $\| A \bigcap G_e \| = \| G_e \| - \| \overline{A} \bigcap G_e \| \ge h(n) - s(p_0(n)) > s(p_0(n))$. □

The following claim is easy to verify from the algorithm, so we do not prove it here.

Claim 7. For some polynomial $p_1(n)$, the algorithm will stop in $p_1(h(n)) + p_1(n)$ steps.

This finishes the proof of Lemma 3.1.

Now we state several consequences of our main lemma, which examine the difference *NP* and other complexity classes.

THEOREM 3.2. *Let H be NP-m-hard. If* $H \triangle A$ *is sparse, then* $NP \subseteq P_{d-\text{maj}}(A)$.

Proof. Let K be *NP-m-complete* and $s(n)$ be a polynomial such that $\text{dist}_{A,H}(n) \le s(n)$. By Lemma 3.1 we get a polynomial $p_0(n)$. Let $h(n) = 2s(p_0(n)) + 1$. From Lemma 3.1(2) it is easy to see that $K \le^P_{d-\text{maj}} A$. So $NP \subseteq P_{d-\text{maj}}(A)$. □

It is well known that the class *coNP* is closed under positive truth-table reductions [11]. Since $\le^P_{d-\text{maj}}$ is a special kind of positive reduction, we have the following result.

COROLLARY 3.3. *Let H be NP-m-hard. If H is coNP-close, then NP = coNP.*

It is easy to see that R (see [2]) is closed under positive truth-table reductions. So we also have the following corollary.

COROLLARY 3.4. *Let H be NP-m-hard. If H is R-close, then NP =R.*

Unfortunately we do not know whether *UP*, *FewP*, or $C_=P$ is closed under $\le^P_{d-\text{maj}}$-reductions. Thus, we cannot obtain the similar results for *UP*, *FewP*, or $C_=P$.

THEOREM 3.5. *Let H be NP-m-hard and* $A \subseteq \Sigma^*$. *If* $\text{dist}_{A,H}(n) = O(\log n)$, *then NP* $\subseteq P_{d-\text{conj}}(A)$.

Proof. Let K be *NP-m-complete*. Since $\| (H \triangle A)^{\le n} \| \le c \cdot \log n + c$ for some $c > 0$, let $s(n) = c \cdot \log n + c$. Now use Lemma 3.1 and let $h(n) = 2s(p_0(n)) + 1$. Clearly $h(n) = O(\log n)$. By Lemma 3.1(2) we have $x \in K \iff$ (The algorithm accepts x) or $(\| G_e \bigcap A \| > s(p_0(n))$ for some $e \le e_x$). Since $\| G_e \bigcap A \| > s(p_0(n)) \iff$ there exists $G((G \subseteq G_e) \wedge (\| G \| = s(p_0(n)) + 1) \wedge (G \subseteq A))$.

So $(\| G_e \bigcap A \| > s(p_0(n))$ for some $e \le e_x) \iff$ there exists $e \le e_x$ there exists $G((G \subseteq G_e) \wedge (\| G \| = s(p_0(n)) + 1) \wedge (G \subseteq A))$.

Since $\| G_e \|$ is $O(\log n)$, the number of subsets of G_e is bounded by a polynomial. Therefore, $K \in P_{d-\text{conj}}(A)$.

Hence $NP \subseteq P_{d-\text{conj}}(A)$. □

COROLLARY 3.6. *Let H be NP-m-hard. If H is* $O(\log n)$*-UP-close, then NP = FewP.*

LEMMA 3.7. $C_=P$ *is closed under* $\le^P_{d-\text{conj}}$*-reductions.*

Proof. Clearly, $A \le^P_{d-\text{conj}} B$ implies that there exists a C such that $A \le^P_{dtt} C$ and $C \le_{ctt} B$. Since $C_=P$ is closed under both $\le^P_{dtt}$- and $\le^P_{ctt}$-reductions (see [4]), it is easy to see that the lemma holds. □

COROLLARY 3.8. *Let H be NP-m-hard. If H is* $O(\log n)$*-*$C_=$ *P-close, then NP* $\subseteq C_=P$.

THEOREM 3.9. *Let H be NP-m-hard and A be a language. If* $H \subseteq A$ *and* $A - H$ *is sparse, then* $NP \subseteq P_{d-\text{conj}}(A)$.

Sketch of Proof. The proof of this theorem is very similar to that of Lemma 3.1, except let $h(n) = s(p_0(n)) + 1$ and replace Claim 4 and Claim 6 by the following Claim 4′ and Claim 6′ respectively. □

Claim 4′. Let $e \le e_x$. If $G_e \bigcap \overline{A} \neq \emptyset$, then $w_{\max}$ is in an interval of $U_e \iff w_{\max}$ is in an interval of V_e.

Proof of Claim 4′. Since $G_e \bigcap \overline{A} \neq \emptyset$ and $H \subseteq A$, we have some interval $[c_i, d_i]$ in V_e such that $f(\langle x, c_{i_1}\rangle) \notin H$. As in Claim 4 $w_{\max}$ is in an interval of $U_e \iff w_{\max}$ belongs to an interval in V_e. □

Claim 6′. $x \in K \iff$ (The algorithm accepts x) or ($G_e \subseteq A$ and $\| G_e \| = h(n)$ for some $e \le e_x$).

From Theorem 3.9 , Lemma 3.6, and Lemma 3.9 we have:

COROLLARY 3.10. *Let H be NP-m-hard. If H is UP-outside-close, then NP = FewP.*

COROLLARY 3.11. *Let H be NP-m-hard. If H is $C_=P$-outside-close, then $NP \subseteq C_=P$.*

We left some open problems:

1. Does $NP \neq coNP(NP \neq R)$ imply that for any *NP-m-hard* set H and $A \in coNP$ (respectively, R), $A \triangle H$ is not in $P_{btt}(sparse)$?

2. Does $FewP \neq NP$ imply that for any *NP-m-hard* set H and $A \in FewP$, $A \triangle H$ is not sparse?

3. Does $UP \neq NP$ imply that for any *NP-m-hard* set H, H is not *UP-inside-close*?

Acknowledgments. We would like to thank Professor Shouwen Tang for his encouragement in this research. The first author also thanks Professor Qiongzhang Li for his encouragement during this research and Mr. Tian Liu for his valuable suggestions for the earlier version of this paper [6]. We thank Mr. Anjiang Ma for his patience in typing this paper and the unknown referees for their suggestions for improving our presentation.

REFERENCES

[1] E. ALLENDER AND R. RUBINSTEIN, *P-printable sets*, SIAM J. Comput., 17 (1988), pp. 1192–1202.

[2] J. BALCÁZAR, J. DIAZ, AND J. GABARRO, *Structural Complexity* I, II, Springer-Verlag, New York, 1988 and 1990.

[3] R. BEIGEL, N. REINGOLD, AND D. SPIELMAN, *PP is closed under intersection*, in Proc. 23rd ACM Symposium on Theory of Computing, New Orleans, 1991, pp. 1–9.

[4] R. BEIGEL, R. CHANG, AND M. OGIWARA, *A Relationship between Difference Hierachies and Relativized Polynomial Hierarchies*, CORNEL/DCS/TR 1184, Cornell University, Dept. of Computer Science, 1991.

[5] B. FU, *On lower bounds of the closeness between complexity classes*, Math. Systems Theory, 26 (1993), pp. 187–202.

[6] ———, *On Closeness of NP-hard sets to coNP*, manuscript, May 1991.

[7] S. HOMER AND L. LONGPRE, *On reductions of NP sets to sparse sets*, in Proc. 6th IEEE Conference on Structural Complexity Theory, Chicago, 1991, pp. 79–88.

[8] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th Symposium on the Theory of Computing, 1980, pp. 302–309.

[9] M. OGIWARA AND O. WATANABE, *On polynomial time bounded truth-table reducibility of NP sets to sparse sets*, SIAM J. Comput., 20 (1991), pp. 471–483.

[10] U. SCHÖNING, *Complete sets and closeness to complexity classes*, Math. Systems Theory, 19 (1986), pp. 29–41.

[11] L. SELMAN, *Reductions on NP and P-selective sets*, Theoret. Comput. Sci., 19 (1982), pp. 287–304.

[12] S. TANG, B. FU, AND T. LIU, *Exponential time and subexponential time sets*, in Proc. 6th IEEE Conference on Structural Complexity Theory, Chicago, 1991, pp. 230–237.

[13] J. TORAN, *Complexity classes defined by counting quantifies*, J. Assoc. Comput. Mach., 38 (1991), pp. 753–754.

[14] L. VALIANT, *The relative complexity of checking and evaluting*, Inform. Process Lett., 5 (1976), pp. 20–23.

[15] K. WAGNER, *The complexity of combinatorial problems with succinct input representation*, Acta Inform., 23 (1986), pp. 325–356.

[16] O. WATANABE, *On polynomial time one-truth-table reducibility to a sparse set*, J. Comput. System Sci., 44 (1992), pp. 500–516.

[17] Y. YESHA, *On certain polynomial-time truth-table reducibilities of complete sets to sparse sets*, SIAM J. Comput., 12 (1983), pp. 411–425.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 261–275, April 1994

004

COMPLEXITY-RESTRICTED ADVICE FUNCTIONS*

JOHANNES KÖBLER† AND THOMAS THIERAUF‡

Abstract. The authors consider uniform subclasses of the nonuniform complexity classes defined by Karp and Lipton [L'Enseign. Math., 28 (1982)] via the notion of advice functions. These subclasses are obtained by restricting the complexity of computing correct advice. Also, the effect of allowing advice functions of limited complexity to depend on the input rather than on the input's length is investigated. Among other results, using the notions described above, new characterizations of (a) $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$, (b) NP with a restricted access to an NP oracle, and (c) the odd levels of the boolean hierarchy are given.

As a consequence, it is shown that every set that is nondeterministically truth-table reducible to SAT in the sense of Rich [J. Comput. System Sci., 38 (1989), pp. 511–523] is already deterministically truth-table reducible to SAT. Furthermore, it turns out that the NP reduction classes of bounded versions of this reducibility coincide with the odd levels of the boolean hierarchy.

Key words. nonuniform complexity classes, advice classes, optimization functions, restricted oracle access, sparse NP sets, relativization, boolean hierarchy, truth-table reducibility

AMS subject classification. 68Q15

1. Introduction.

In their fundamental paper, Karp and Lipton [23] introduced the notion of advice functions and investigated nonuniform complexity classes which they denoted by $\mathcal{C}/\mathcal{F}$, where $\mathcal{C}$ is a class of sets and $\mathcal{F}$ is a class of (advice) functions. A typical class is P/poly, where poly is the set of polynomially length bounded functions. The interest in P/poly stems from the fact that it consists exactly of the languages that can be computed by polynomially size-bounded circuits [34].

Intuitively, a set A is in $\mathcal{C}/\mathcal{F}$, if A can be solved by a machine of type $\mathcal{C}$ that gets, in addition to the input x, the advice $f(x)$, where f is a function in $\mathcal{F}$ depending only on the length of x. Many researchers have considered nonuniform classes where the function class $\mathcal{F}$ is defined by a quantitative length restriction such as poly and log (see, e.g., [3], [5], [23], [36]). Note that for such $\mathcal{F}$ there are nonrecursive functions in $\mathcal{F}$, and therefore $\mathcal{C}/\mathcal{F}$ contains nonrecursive languages.

Here, we consider uniform language classes obtained by imposing complexity bounds on the advice functions. Note that Kämper [22] investigates refinements of the original $\mathcal{C}/\mathcal{F}$ definition by delimiting the complexity of proof sets, i.e., special sets of correct advice. In contrast to this, we directly bound the complexity of computing correct advice. With this concept, we are able to show characterizations as well as finer distinctions of several complexity classes. For example, we show that the class $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ coincides with the class $\mathrm{NP}/\mathrm{OptP}[O(\log n)]$, a subclass of NP/ log, where correct advice is computable by an OptP function [29], i.e.,

$$\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} = \mathrm{NP}/\mathrm{OptP}[O(\log n)]. \tag{1.1}$$

One can interpret equality (1.1) as stating that (exactly) the languages in $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ can be computed in the following way: on input x of length n, at first an $\mathrm{OptP}[O(\log n)]$ precomputation takes place that gets as input only 1^n. The (logarithmically length-bounded) output

*Received by the editors June 12, 1991; accepted for publication (in revised form) October 16, 1992.

†Abteilung für Theoretische Informatik, Universität Ulm, Oberer Eselsberg, D-89069 Ulm, Germany. The work of this author was supported in part by the Deutscher Akadeuinscher Austauschlienst through Acciones Integradas 1991, 313-AI-e-es/zk.

‡Abteilung für Theoretische Informatik, Universität Ulm, Oberer Eselsberg, D-89069 Ulm, Germany. This author's work was done in part while visiting the University of Rochester and was supported in part by Deutsche Forschunsgemeinschaft Postdoctoral Stipend Th 472/1-1, National Science Foundation grant CCR-8957604, and the Deutscher Akademischer Austauschlienst through Acciones Integradas 1991, 313-AI-e-es/zk.

of this precomputation is then passed along with x to the subsequent NP computation, which decides the membership of x.

Motivated by the relativized separation of P and NP of Baker, Gill, and Solovay [2] (exploiting the fact that an NP oracle machine can ask superpolynomially many queries), Book, Long, and Selman [11] introduced restricted relativizations of NP by bounding the number of oracle queries in various ways. Subsequently, Long [32] investigated the relationship between restricted access of nondeterministic machines to an oracle and full access to a sparse oracle set. Let NP_R^A be the class of all languages whose membership in NP^A is witnessed by an oracle machine such that the number of potential oracle queries in A (asked on some oracle) is polynomially bounded. From this definition, it is clear that $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ is contained in $\mathrm{NP}_R^{\mathrm{NP}}$. Since also coNP is contained in $\mathrm{NP}_R^{\mathrm{NP}}$, $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ and $\mathrm{NP}_R^{\mathrm{NP}}$ are different unless the polynomial hierarchy collapses [21]. By considering the proof of equality (1.1), we will see that if we let the $\mathrm{OptP}[O(\log n)]$ advice function depend not only on the length of the input but on the input itself, we get a characterization of $\mathrm{NP}_R^{\mathrm{NP}}$. This leads us to define the class $\mathcal{C}//\mathcal{F}$, which is defined in the same way as $\mathcal{C}/\mathcal{F}$ but with the advice functions depending on the input. Thus, we obtain the following characterization of $\mathrm{NP}_R^{\mathrm{NP}}$:

$$(1.2) \qquad \mathrm{NP}_R^{\mathrm{NP}} = \mathrm{NP}//\mathrm{OptP}[O(\log n)].$$

The characterizations (1.1) and (1.2) give insight into the difference between restricted access to NP oracles and full access to sparse NP sets.

It seems that the notion of $\mathcal{C}//\mathcal{F}$ is an appropriate concept for studying different kinds of truth-table reducibilities. Let χ_k^{SAT} be the k-ary characteristic function of SAT. Then $\mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ is the class of sets that are k-truth-table reducible to some NP set. It is known that these classes are interleaved with the levels of the boolean hierarchy: $\mathrm{NP}(k) \subseteq \mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP} \subseteq \mathrm{NP}(k+1)$ for all $k \geq 1$ [28]. Since $\mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ is closed under complementation, these classes are all different unless the boolean hierarchy collapses.

$\mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ is the class of sets that are k-truth-table reducible to some NP set, where the evaluator is an NP machine. These classes turn out to coincide with the odd levels of the boolean hierarchy, giving an interesting characterization of the levels of the boolean hierarchy in terms of reduction classes,

$$\mathrm{NP}(2k+1) = \mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}.$$

Furthermore, we show that $\mathrm{NP}(2k+1) = \mathrm{NP}_{k\text{-tt}}^{\mathrm{NP}}$, where $\mathrm{NP}_{k\text{-tt}}^{\mathrm{NP}}$ is the class of sets that are nondeterministically k-truth-table reducible to a set in NP in the sense of [35][1], whereas in the unbounded case all sets nondeterministically truth-table reducible to SAT are already deterministically truth-table reducible to SAT, i.e., $\mathrm{NP}_{\mathrm{tt}}^{\mathrm{NP}} = \mathrm{P}_{\mathrm{tt}}^{\mathrm{NP}}$. The latter result also holds for the strong nondeterministic truth-table reducibility $\leq_{\mathrm{tt}}^{\mathrm{SN}}$ introduced by Long [31], i.e., we show that $\{A \mid A \leq_{\mathrm{tt}}^{\mathrm{SN}} \mathrm{SAT}\} = \{A \mid A \leq_{\mathrm{tt}}^{\mathrm{P}} \mathrm{SAT}\}$.

The paper is organized as follows. Section 2 introduces notation and gives basic definitions. In §3 we prove the aforementioned characterizations of $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ and $\mathrm{NP}_R^{\mathrm{NP}}$ and show that changing from $\mathrm{OptP}[O(\log n)]$ to the larger function class FewOptP (containing all functions whose membership in OptP is witnessed by an NP transducer that generates only polynomially many different outputs) does not increase the power of $\mathrm{NP}/\mathrm{OptP}[O(\log n)]$ and $\mathrm{NP}//\mathrm{OptP}[O(\log n)]$.

In §4 we separate some of these complexity classes in relativized world, the main result is a separation of $\mathrm{P}/\mathrm{OptP}[O(\log n)]$ and $\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$.

[1] By requiring the NP generator to be single valued, Rich [35] has modified the nondeterministic truth-table reducibility originally defined by Ladner, Lynch, and Selman [30].

In §5 we give several characterizations of certain levels of the boolean hierarchy in terms of various complexity restricted advice function classes.

2. Preliminaries and notation. All languages considered here are over the alphabet $\Sigma = \{0, 1\}$. For a string $x \in \Sigma^*$, $|x|$ denotes its length. We assume the existence of a pairing function $\langle \cdot, \cdot \rangle : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ that is computable in polynomial time and has inverses also computable in polynomial time. $\langle \cdot, \cdot \rangle$ can be extended to encode finite sequences $(x_1, \dots, x_k)$ of strings into a string $\langle x_1, \dots, x_k \rangle \in \Sigma^*$. For a set A, $|A|$ denotes its cardinality. The complement $\Sigma^* - A$ of A is denoted by $\overline{A}$. $A^{\leq n}$ is the set of all strings in A of length less than or equal to n.

A language S is *sparse* if there is a polynomial p such that for all n the number of words in S up to length n is at most $p(n)$. Let SPARSE be the class of all sparse languages. A set T is *tally* if T is a subset of 1^*. Let TALLY be the class of all tally sets.

We assume that the reader is familiar with (nondeterministic, polynomial-time bounded, oracle) Turing machines and complexity classes (see [4], [36]). FP is the class of functions computable by a deterministic polynomial-time bounded Turing transducer. An *NP transducer* is a nondeterministic polynomial-time bounded Turing machine T that on every branch either accepts and writes a binary number on its output tape or rejects. The set of outputs generated by T on input x is denoted by $\mathrm{out}_T(x)$.

Krentel [29] defines an *NP metric Turing machine* to be an NP transducer that accepts on every branch. For an NP metric Turing machine T and an input $x \in \Sigma^*$ let $\max_T(x)$ $[\min_T(x)]$ be the maximum [minimum] output generated by T on input x on any accepting computation of T. The class OptP [29] of optimization functions is defined as

$$\mathrm{OptP} = \{\max_T, \min_T \mid T \text{ is an NP metric Turing machine}\}.$$

For a class $\mathcal{R}$ of functions on the natural numbers (called restricting functions), we define the subclass

$$\mathrm{OptP}[\mathcal{R}] = \{f \in \mathrm{OptP} \mid \exists r \in \mathcal{R} \, \forall x \in \Sigma^* : |f(x)| \leq r(|x|)\}$$

containing all optimization functions $f \in \mathrm{OptP}$ such that the length of $f(x)$ in binary representation is bounded by a function from $\mathcal{R}$.

$\mathrm{P}^{\mathrm{NP}[\mathcal{R}]}$ denotes the class of sets whose membership in P^{NP} can be witnessed by an oracle machine M making for some $r \in \mathcal{R}$ at most $r(n)$ many queries on inputs of length n. In the case that $\mathcal{R}$ is a singleton set $\{r\}$ we simply write $\mathrm{OptP}[r]$ and $\mathrm{P}^{\mathrm{NP}[r]}$, respectively. Throughout the paper we assume that for every restricting function r the function $x \mapsto r(|x|)$ is computable in polynomial time.

Karp and Lipton [23] introduced the notion of advice functions in order to define nonuniform complexity classes. For a class $\mathcal{C}$ of sets and a class $\mathcal{F}$ of functions from Σ^* to Σ^* let $\mathcal{C}/\mathcal{F}$ be the class of sets A such that there is a set $B \in \mathcal{C}$ and a function $h \in \mathcal{F}$ such that for all $x \in \Sigma^*$,

$$x \in A \Leftrightarrow \langle x, h(1^{|x|}) \rangle \in B.$$

Note that the advice function h depends only on the length of x. By canceling this restriction we obtain the class $\mathcal{C}//\mathcal{F}$ of all sets A such that there is a set $B \in \mathcal{C}$ and a function $h \in \mathcal{F}$ such that for all $x \in \Sigma^*$,

$$x \in A \Leftrightarrow \langle x, h(x) \rangle \in B.$$

By definition, $\mathcal{C}/\mathcal{F}$ is a subset of $\mathcal{C}//\mathcal{F}$ for each class of sets $\mathcal{C}$ and each class of functions $\mathcal{F}$, which fulfills the condition that if $h \in \mathcal{F}$ then also $x \mapsto h(1^{|x|}) \in \mathcal{F}$. Special advice function classes considered in the literature are poly $= \{h : \Sigma^* \rightarrow \Sigma^* \mid$ there exists a polynomial p such that for all x, $|h(x)| \leq p(|x|)\}$ and log $= \{h : \Sigma^* \rightarrow \Sigma^* \mid |h(x)| = O(\log(|x|))\}$.

3. $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ versus $\mathrm{NP}_R^{\mathrm{NP}}$. In this section we show that the class $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ can be characterized as the class $\mathrm{NP}/\mathrm{OptP}[O(\log n)]$, i.e., the class of sets that are accepted by an NP machine with advice of a logarithmically length-bounded OptP function. Further, it turns out that the related class $\mathrm{NP}_R^{\mathrm{NP}}$ (see definition below) coincides with $\mathrm{NP}//\mathrm{OptP}[O(\log n)]$. For the latter two classes we can show that they are also equal to $\mathrm{P}//\mathrm{OptP}[O(\log n)]$, which, by a result of Krentel [29], is identical to $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$.

DEFINITION 3.1 [11]. *For any oracle Turing machine M and any string $x \in \Sigma^*$ let $Q(M, A, x)$ be the set of all oracle queries that M may ask on input x using oracle A, i.e., the set of all strings $y \in \Sigma^*$ such that in some computation of M on input x under oracle A the oracle is queried about y. $Q(M, x)$ is the set of all oracle queries of M on input x using any oracle, i.e., $Q(M, x) = \bigcup_{A\subseteq\Sigma^*} Q(M, A, x)$.*

For any set A, $\mathrm{NP}_R(A)$ is the class of sets $L \in \mathrm{NP}(A)$ whose membership is witnessed by a machine M such that the number of potential oracle queries in A is polynomially bounded, i.e., there exists a polynomial p such that $|Q(M, x) \cap A| \leq p(|x|)$ for all x.

Our first theorem states that if a language L is accepted by an NP oracle machine M using an NP oracle A in such a way that the number of potential oracle queries that are in A is polynomially bounded, then L is in $\mathrm{NP}//\mathrm{OptP}[O(\log n)]$, i.e., membership in L can be tested by an NP machine which gets along with the input the value of an $\mathrm{OptP}[O(\log n)]$ function. In the special case that A is sparse this containment can be strengthened to $\mathrm{NP}/\mathrm{OptP}[O(\log n)]$, i.e., for all inputs of the same length the advice function yields the same result. The proof is by a census argument similar to that used by Hemachandra [18] and Kadin [21].

THEOREM 3.2. (i) $\mathrm{NP}_R^{\mathrm{NP}} \subseteq \mathrm{NP}//\mathrm{OptP}[O(\log n)]$,

(ii) $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} \subseteq \mathrm{NP}/\mathrm{OptP}[O(\log n)]$.

Proof. Let $L = L(M, A)$ for an NP machine M and an oracle A in NP, and let p be a polynomial that bounds the running time of M.

To show (i) let r be a polynomial such that $|Q(M, x) \cap A| \leq r(|x|)$ for all x. An NP machine knowing the size of the set $Q(M, x) \cap A$ can guess this set (note that it is an NP problem to decide for two given strings x and y whether y is in $Q(M, x) \cap A$). Define the function

$$h(x) = |Q(M, x) \cap A|$$

and the set

$$B = \{\langle x, k\rangle \mid \exists X \subseteq Q(M, x) \cap A : |X| = k \text{ and } x \in L(M, X)\}.$$

Then $B \in \mathrm{NP}$ and $h \in \mathrm{OptP}[O(\log |x|)]$, since $h(x)$ is the maximum output of the following algorithm:

```
On input $x$ guess $k \leq r(|x|)$ and $x_1 < \cdots < x_k \in \Sigma^{\leq p(|x|)}$;
if $x_1, \ldots, x_k \in Q(M, x) \cap A$, then output $k$, else output 0.
```

Now, it holds for all $x \in \Sigma^*$ that $x \in L$ if and only if $\langle x, h(x)\rangle \in B$. Therefore, L is in $\mathrm{NP}//\mathrm{OptP}[O(\log n)]$.

For (ii) let A be a sparse set, and let r be a polynomial such that $|A^{\leq p(n)}| \leq r(n)$ for all n. Define the function

$$h(x) = |A^{\leq p(|x|)}|$$

and the set

$$B = \{\langle x, k\rangle \mid \exists X \subseteq A^{\leq p(|x|)} : |X| = k \text{ and } x \in L(M, X)\}.$$

By an argument similar to that in the proof of (i), $x \in L$ if and only if $\langle x, h(1^{|x|})\rangle \in B$. This shows that L is in NP/OptP[$O(\log n)$]. □

Combining Theorem 3.2 (ii) with the result of Balcázar and Schöning [5] that NP/log ∩ coNP $\subseteq$ $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ (see also [3]), it follows that for every coNP set in NP/log correct advice can already be computed by an OptP function.

COROLLARY 3.3. NP/log ∩ coNP $=$ NP/OptP[$O(\log n)$] ∩ coNP.

To show the reverse containments of Theorem 3.2, we make use of the following lemma. It states that every OptP function h can be computed by a deterministic polynomial-time oracle machine by asking $|h(x)|$ many queries to an NP oracle.

LEMMA 3.4 [29]. OptP[r] $\subseteq$ $\mathrm{FP}^{\mathrm{NP}[r]}$ *for any restricting function r.*

COROLLARY 3.5. (i) $\mathrm{NP}_R^{\mathrm{NP}} = $ NP//OptP[$O(\log n)$],

(ii) $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} = \mathrm{NP}^{\mathrm{NP}\cap\mathrm{TALLY}} =$ NP/OptP[$O(\log n)$].

Proof. By Theorem 3.2, it only remains to show the inclusions from right to left.

To show (i), let L be in NP//OptP[$O(\log n)$] via an NP machine N and an optimization function h. Then L can be accepted by an NP machine M that computes deterministically by binary search the value of h according to Lemma 3.4 asking $O(\log n)$ many queries to an NP oracle and then simulates N without asking further oracle queries. Since $Q(M, x)$ is polynomially bounded, it follows that L is in $\mathrm{NP}_R^{\mathrm{NP}}$.

If h is a function that depends only on the length of its argument, then $h(x)$ can be computed by binary search using the tally NP set $T = \{1^{\langle n,k\rangle} \mid k \leq h(1^n)\}$. This proves (ii). □

Note that the above proof shows that every language in NP//OptP[$O(\log n)$] (and thus in $\mathrm{NP}_R^{\mathrm{NP}}$) can in fact be accepted by an NP oracle machine M such that $Q(M, x)$ is polynomially bounded.

In the next lemma, we show that an NP computation getting along with the input the value function $h(x)$ of an OptP can be transformed into a P computation on the cost of adding one bit to $h(x)$. Note that this bit actually depends on x even if $h(x)$ only depends on the length of x.

LEMMA 3.6. NP//OptP[r] $\subseteq$ P//OptP[$r + 1$] *for any function r.*

Proof. Let L be in NP//OptP[r], witnessed by an NP set B and an OptP[r] function $h = \max_T$ for some NP metric machine T. Define the OptP[$r + 1$] function

$$h'(x) = \begin{cases} h(x)1 & \text{if } \langle x, h(x)\rangle \in B, \\ h(x)0 & \text{otherwise.} \end{cases}$$

Then it holds for all x that $\langle x, h(x)\rangle \in B \Leftrightarrow \langle x, h'(x)\rangle \in B'$, where the set $B' = \{\langle x, k\rangle \mid k \text{ is odd}\}$ is in P. The case that $h = \min_T$ can be proved analogously. □

Combining Corollary 3.5(i) and Lemma 3.6 we obtain a further characterization of the class $\mathrm{NP}_R^{\mathrm{NP}}$ and that it is closed under complementation. Note that P//OptP[$O(\log n)$] $=$ $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$ [29].

COROLLARY 3.7. $\mathrm{NP}_R^{\mathrm{NP}} = $ P//OptP[$O(\log n)$].

COROLLARY 3.8. $\mathrm{NP}_R^{\mathrm{NP}}$ *is closed under complementation.*

Remark 3.9. The results stated in Corollary 3.5 can be extended to the classes of the polynomial-time hierarchy [37]. In order to do so, we define restricted relativizations of the Σ-levels of the polynomial hierarchy. $\Sigma_{k,R}^{\mathcal{C}}$ is the class of all sets L accepted by a k-alternating polynomial-time Turing machine [16] using an oracle A from $\mathcal{C}$ such that $|Q(M, x) \cap A|$ is polynomially bounded. Then, the results stated in Corollary 3.5 can be extended to

$$\begin{aligned} \Sigma_k^{\Sigma_k\cap\mathrm{SPARSE}} &= \Sigma_k/\mathrm{Opt}\,\Sigma_{k-1}[O(\log n)], \\ \Sigma_{k,R}^{\Sigma_k} &= \Sigma_k//\mathrm{Opt}\,\Sigma_{k-1}[O(\log n)] = \mathrm{P}^{\Sigma_k[O(\log n)]}, \end{aligned}$$

where Opt $\mathcal{C}$ is the class of optimization functions computable by an NP transducer using some oracle in the class $\mathcal{C}$. Since $\Sigma_k/\mathrm{Opt}\,\Sigma_{k-1}[O(\log n)]$ is included in $\mathrm{P}^{\Sigma_k[O(\log n)]}$, this sharpens the recent result in [13] that $\Sigma_k^{\Sigma_k \cap \mathrm{SPARSE}} \subseteq \mathrm{P}^{\Sigma_k[O(\log n)]}$.

Remark 3.10. The advice (even depending on the input) provided by an OptP[$O(\log n)$] function does not increase the power of the probabilistic class PP: PP//OptP[$O(\log n)$] = PP. This follows from the result by Toda [40] that $\mathrm{PP}_R^{\mathrm{NP}} = \mathrm{PP}$, since PP//OptP[$O(\log n)$] coincides with the class $\mathrm{PP//FP}^{\mathrm{NP}[O(\log n)]}$ (see Lemma 3.4) that is clearly contained in $\mathrm{PP}_R^{\mathrm{NP}}$.

Next, we consider uniform subclasses of P/log and P/poly. Whereas the proof of Corollary 3.5(ii) also yields the inclusion of P/OptP[$O(\log n)$] in $\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$, the census technique of Theorem 3.2 cannot be applied to obtain the reverse containment. The next theorem is proved by constructing (long enough initial segments of) a sparse NP set by an OptP computation. The underlying technique was used by Mahaney [33] to show that $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} \subseteq \mathrm{P}^{\mathrm{NP}}$.

THEOREM 3.11. $\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}} \subseteq \mathrm{P/OptP}$.

Proof. Let $L = L(M, S)$ for a P machine M and a sparse NP set S. Let p and r be polynomials such that p bounds the running time of M and $|S^{\le n}| \le r(n)$. Define $h(x) = \langle S^{\le p(|x|)}\rangle$. Then $h \in$ OptP, since $h(x)$ is the maximum output of the following algorithm:

```
On input $1^n$ guess $k \le r(p(n))$ and $x_1 < \cdots < x_k \in \Sigma^{\le p(|x|)}$;
if $x_1, \ldots, x_k \in S$, then output $\langle x_1, \ldots, x_k\rangle$, else output 0.
```

Now the computation of M using oracle S on input x can be simulated by a P machine answering oracle questions according to the set $h(1^{|x|})$. □

Let FewOptP be the class of functions $f \in$ OptP computed by an NP transducer that produces at most a polynomial number of different outputs. Clearly, OptP[$O(\log n)$] $\subseteq$ FewOptP, and obviously, this is a proper inclusion.

However, as shown by the next theorem, the classes NP/OptP[$O(\log n)$] and NP//OptP[$O(\log n)$] remain unchanged when the function class OptP[$O(\log n)$] is replaced by the larger class FewOptP.

THEOREM 3.12. (i) NP//FewOptP = P//FewOptP = P//OptP[$O(\log n)$],

(ii) NP/FewOptP = NP/OptP[$O(\log n)$].

Proof. Let L be a set in NP//FewOptP via $A \in$ NP and $f \in$ FewOptP. Let T be an NP metric machine for f, i.e., $f = \max_T$ (the proof for $f = \min_T$ is similar), and the number of different outputs of T is polynomially bounded. Define the function

$$h(x) = |\mathrm{out}_T(x)|$$

and the set

$$B = \{\langle x, m\rangle \mid \exists z_1 < \cdots < z_m \in \mathrm{out}_M(x) : \langle x, z_m\rangle \in A\}.$$

It is easy to see that $h \in$ OptP[$O(\log n)$] and $B \in$ NP. Now x is in L if and only if $\langle x, h(x)\rangle$ is in B, and therefore L is in NP//OptP[$O(\log n)$] = P//OptP[$O(\log n)$]. The latter equality follows from Corollaries 3.5(i) and 3.7. The proof of (ii) is analogous, we only have to replace $\mathrm{out}_T(x)$ by $\mathrm{out}_T(1^{|x|})$. □

The technique used in the previous proof cannot be applied to show that the classes P/OptP[$O(\log n)$] and P/FewOptP are equal. However, the proof of P/OptP[$O(\log n)$] $\subseteq \mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$ (using binary search, see the proof of Corollary 3.5(ii) can be refined to show the following theorem. It states that a set in P/FewOptP can be decided in polynomial time by querying a sparse NP oracle (polynomially often).

THEOREM 3.13. $\mathrm{P/FewOptP} \subseteq \mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}}$.

Proof. Let f be in FewOptP, and let T be an NP transducer computing f. Using the sparse NP set

$$S = \{\langle 1^n, m, i, z\rangle \mid \exists z_1 < \cdots < z_m \in \mathrm{out}_T(1^n)\ \exists z' :\ z z' = z_i\}$$

as oracle, $f(x)$ can be computed in polynomial time by determining first $|\mathrm{out}_T(1^n)|$ and then applying a prefix search to find the optimum value in $\mathrm{out}_T(1^n)$. □

The known relationships of the language classes considered in this section are summarized in Fig. 3.1.

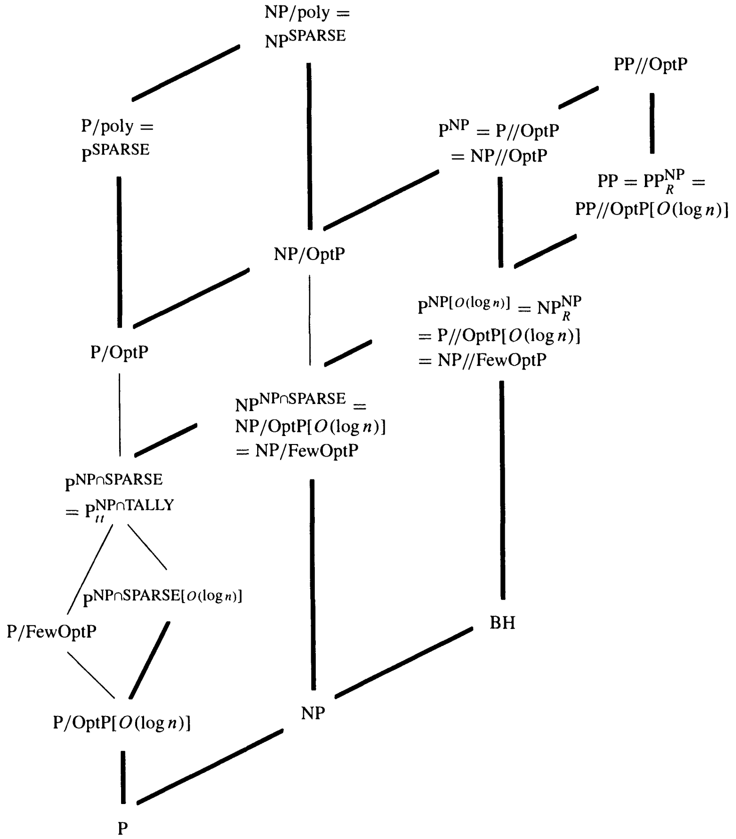

$\mathrm{NP/poly} = \mathrm{NP}^{\mathrm{SPARSE}}$
PP//OptP
$\mathrm{P/poly} = \mathrm{P}^{\mathrm{SPARSE}}$
$\mathrm{P}^{\mathrm{NP}} = \mathrm{P//OptP} = \mathrm{NP//OptP}$
$\mathrm{PP} = \mathrm{PP}^{\mathrm{NP}}_{R} = \mathrm{PP//OptP}[O(\log n)]$
NP/OptP
$\mathrm{P}^{\mathrm{NP}[O(\log n)]} = \mathrm{NP}^{\mathrm{NP}}_{R} = \mathrm{P//OptP}[O(\log n)] = \mathrm{NP//FewOptP}$
P/OptP
$\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} = \mathrm{NP/OptP}[O(\log n)] = \mathrm{NP/FewOptP}$
$\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}} = \mathrm{P}^{\mathrm{NP}\cap\mathrm{TALLY}}_{tt}$
$\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$
P/FewOptP
BH
$\mathrm{P/OptP}[O(\log n)]$
NP
P

FIG. 3.1. *Inclusion structure of some considered complexity classes; thick lines indicate that there are relativized separations* (*see* §4).

4. Relativized separations. Since Baker, Gill, and Solovay [2] separated P from NP relative to some oracle, relativizations have been an important subject in complexity theory. In this section, we discuss which of the inclusions in Fig. 3.1 are strict, at least in some relativized world.

Since there are nonrecursive sets in P/poly and in NP/poly, these two classes are clearly different from all other (recursive) classes considered here. Whether there are any other strict inclusions in the diagram of Fig. 3.1 is not known. For some of the inclusions, however, the question whether they are proper can be linked to central open problems in complexity theory.

For example, by the result of Karp, Lipton, and Sipser (see [23]) that $\mathrm{NP} \subseteq \mathrm{P/poly}$ implies the collapse of the polynomial hierarchy to its second level, it follows that if $\mathrm{PH} \neq \Sigma_2$, then

NP is not contained in any of the classes on the left column of Fig. 3.1. Since this holds in all relativized worlds, and since there exists an oracle separating PH from Σ_2 [24], it follows that relative to this oracle all the inclusions between the first and the second column are proper.

Similarly, using the result of Kadin [21] that coNP $\subseteq$ $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}}$ implies PH $=$ $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$, it follows that if PH $\neq$ $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$, then $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{SPARSE}} \neq \mathrm{P}^{\mathrm{NP}[O(\log n)]}$. Since, as it is easily seen, the inclusion coNP $\subseteq$ NP/OptP implies PH $=$ P^{NP}, we can state the following theorem.

THEOREM 4.1. PH $\neq \mathrm{P}^{\mathrm{NP}} \Rightarrow$ NP/OptP $\neq$ P//OptP.

Furthermore, by the recent result of Toda [39] that PH $\subseteq \mathrm{P}^{\mathrm{PP}}$, it follows that $\mathrm{P}^{\mathrm{NP}[O(\log n)]} \neq$ PP and $\mathrm{P}^{\mathrm{NP}} \neq$ PP//OptP unless PH $= \mathrm{P}^{\mathrm{NP}}$.

Beigel [7] constructed an oracle A such that $\mathrm{P}^{\mathrm{NP}^A} - \mathrm{PP}^A \neq \emptyset$. Since $\mathrm{P}^{\mathrm{NP}[O(\log n)]} \subseteq$ PP [9], oracle A also separates $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$ and P^{NP} (for a direct proof see [14]).

Cai et al. [15] showed the existence of an oracle A such that relative to A the boolean hierarchy is infinite, i.e., $\forall k : \mathrm{NP}^A(k) \neq \mathrm{coNP}^A(k)$. In fact, Cai et al. construct the oracle A in such a way that, for all k, some *tally* test language $L_k(A)$ is in $\mathrm{coNP}^A(k) - \mathrm{NP}^A(k)$. Because it holds for every oracle set B that

$$\mathrm{NP}^B(2^k - 1) \cup \mathrm{coNP}^B(2^k - 1) \subseteq \mathrm{P//OptP}^B[k] \subseteq \mathrm{NP}^B(2^k)$$

[8], [28], [43], it follows that $L_{2^k-1}(A) \in \mathrm{P//OptP}^A[k] \cap \mathrm{TALLY} \subseteq \mathrm{P/OptP}^A[k]$, i.e.,

$$\exists A \ \forall k \geq 1 : \mathrm{P/OptP}^A[k] - \mathrm{NP}^A(2^k - 1) \neq \emptyset.$$

Since P/OptP$[k]$ is contained in the 2^kth level of the boolean hierarchy, this result is optimal.

Clearly, if the boolean hierarchy is proper, it does not have complete sets. Since the class P/OptP$[O(\log n)]$ has complete sets, it is not contained in BH in any relativized world where the boolean hierarchy is infinite, i.e.,

$$\exists A : \mathrm{P/OptP}^A[O(\log n)] - \mathrm{BH}^A \neq \emptyset.$$

The main result in this section is a separation of the classes P/OptP$[O(\log n)]$ and $\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$. In fact, we show that for any fixed polynomial q there is a relativization such that NP contains sparse sets that are not in the nonuniform class P/q (defined as $\mathrm{P}/\{h \mid |h(x)| \leq q(|x|)\}$).

THEOREM 4.2. *For every polynomial q there exists a set A such that*

$$(\mathrm{NP}^A \cap \mathrm{SPARSE}) - \mathrm{P}^A/q \neq \emptyset.$$

Proof. For an arbitrary set A we define a sparse set $L(A) \in \mathrm{NP}^A$ as follows. For a given n and a suitable chosen function $l(n)$, we partition the $2^{l(n)}$ words of length $l(n)$ into $q(n) + 1$ intervals (with respect to the lexicographic ordering) $I_1^{l(n)}, \ldots, I_{q(n)+1}^{l(n)}$ such that

$$|I_k^{l(n)}| \geq \left\lfloor \frac{2^{l(n)}}{q(n)+1} \right\rfloor \quad \text{for } k = 1, \ldots, q(n) + 1.$$

For each interval containing a word in A, we put a word into $L(A)$: let $w_1^n, w_2^n, \ldots$ be an enumeration of Σ^n in lexicographic order, and let $l(n) = n + q(n)$. Define the NP^A set

$$L(A) = \{w_k^n \mid n \geq 1, \ 1 \leq k \leq q(n) + 1 \text{ and } I_k^{l(n)} \cap A \neq \emptyset\}.$$

Clearly, there are at most $q(n) + 1$ words of length n in A, i.e., $L(A)$ is sparse. Now we construct a set A in stages such that $L(A) \notin \mathrm{P}/q$. Let $M_1, M_2, \ldots$ be an enumeration of all polynomial-time bounded Turing machines with running times $p_1, p_2, \ldots$, respectively.

Stage 0. $A := \emptyset$; $n_0 := \min\{n \mid \forall m \geq n : \ q(m) < 2^m\}$.

Stage $s \geq 1$. Choose n_s minimal such that $n_s > \max\{p_i(n_{s-1}) \mid i < s\}$ and $2^{n_s} > 2\, p_s(n_s)\,(q(n_s)+1)^2$.

The algorithm in Fig. 4.1 determines the words of length $l(n_s)$ that are included in A. This is done by diagonalizing against machine M_s and all potential advice for M_s on an input of length n_s.

```
ADVICE := $\Sigma^{\leq q(n_s)}$;
(* ADVICE contains all potential advice against that we have to diagonalize *)
QUERY := $\emptyset$;
(* In QUERY we freeze the oracle queries of $M_s$ during the construction *)
for $k := 1$ to $q(n_s) + 1$ do
  ACC := $\{a \in$ ADVICE $\mid M_s^A(w_k^{n_s}, a)$ accepts $\}$;
  REJ := ADVICE − ACC;
  if |ACC| ≥ |REJ| then
    (* $I_k^{l(n_s)} \cap A$ remains empty, i.e., no word in ACC is advice for $w_k^{n_s}$ *)
    ADVICE := REJ;
    QUERY := QUERY $\cup \bigcup_{a \in \mathrm{ACC}} Q(M_s, A, \langle w_k^{n_s}, a\rangle)$ ;
  else
    (* put a word in $I_k^{l(n_s)} \cap A$, i.e., no word in REJ is advice for $w_k^{n_s}$ *)
    ADVICE := ACC;
    QUERY := QUERY $\cup \bigcup_{a \in \mathrm{REJ}} Q(M_s, A, \langle w_k^{n_s}, a\rangle)$ ;
    choose a $y \in I_k^{l(n_s)}$−QUERY;
    $A := A \cup \{y\}$
  end (* if *)
end (* for *).
```

FIG. 4.1. *Algorithm used in the proof of Theorem* 4.2.

Let M be any P machine. We show that M, taking advice of any q-length bounded function, does not accept $L(A)$. Let s be an index such that $M = M_s$. There are $2^{q(n_s)+1} - 1$ potential words as advice for M_s on inputs of length n_s (that are stored in ADVICE). Each execution of the for-loop diagonalizes against at least half of the possible advice for M_s. Since $\log(2^{q(n_s)+1} - 1) \leq q(n_s) + 1$, ADVICE becomes empty at the end of the algorithm. The construction further guarantees that for every advice a, $|a| \leq q(n_s)$, there exists a $k \leq q(n_s)+1$ such that

$$\langle w_k^{n_s}, a\rangle \in L(M_s, A) \Leftrightarrow w_k^{n_s} \notin L(A).$$

Therefore, it suffices to show that the algorithm can always find a $y \in I_k^{l(n_s)}$– QUERY. In every execution of the for-loop and for every advice no more than $p_s(n_s)$ words are added to the set QUERY, i.e.,

$$\mid \text{QUERY} \mid \leq (q(n_s) + 1)\, 2^{q(n_s)+1}\, p_s(n_s).$$

Thus, we have for $1 \leq k \leq q + 1$,

$$\begin{aligned} |I_k^{l(n_s)} - \text{ QUERY} \mid &\geq |I_k^{l(n_s)}| - \mid \text{QUERY} \mid \\ &\geq \left\lfloor \frac{2^{n_s+q(n_s)}}{q(n_s)+1} \right\rfloor - (q(n_s)+1)\, 2^{q(n_s)+1}\, p_s(n_s) \end{aligned}$$

$$\begin{aligned} &\geq \frac{2^{q(n_s)+1}(2^{n_s-1}-(q(n_s)+1)^2\,p_s(n_s))}{q(n_s)+1}-1 \\ &\geq \frac{2^{q(n_s)+1}}{q(n_s)+1}-1 \quad \text{[by choice of } n_s\text{]} \\ &> 0. \quad \square \end{aligned}$$

COROLLARY 4.3. $\exists A : (\mathrm{NP}^A \cap \mathrm{SPARSE}) - \mathrm{P}^A/\log \neq \emptyset$.

Using a "Kolmogorov-argument", Corollary 4.3 was already shown by Hemachandra [19]. An immediate consequence of Corollary 4.3 is the existence of an oracle separating $\mathrm{P}/\mathrm{OptP}[O(\log n)]$ and $\mathrm{P}^{\mathrm{NP}\cap\mathrm{SPARSE}[O(\log n)]}$.

COROLLARY 4.4. $\exists A : \mathrm{P}^A/\mathrm{OptP}^A[O(\log n)] \neq \mathrm{P}^{\mathrm{NP}^A\cap\mathrm{SPARSE}[O(\log n)]}$.

5. Bounded advice versus the boolean hierarchy. The boolean hierarchy is the smallest class of sets that contains NP and is closed under union, intersection, and complementation. The levels of the boolean hierarchy are obtained by applying a fixed number of boolean operations to NP sets. In this section, we give several characterizations of the odd levels of the boolean hierarchy. First, we show that NP machines that get as advice the value of the k-ary characteristic function χ_k^{SAT} of SAT, where χ_k^{SAT} is evaluated on a k-tuple that is computed from the input by an FP function, accept exactly the languages in the $(2k+1)$th level of the boolean hierarchy. The same is true if the advice consists only of the information of how many out of k words that are produced from the given input by an FP function are in SAT.

Cai et al. [15] give several characterizations of the boolean hierarchy; we take the following one.

DEFINITION 5.1. *A set L is in the* kth level $\mathrm{NP}(k)$ of the boolean hierarchy *if there exist sets $L_1, \dots, L_k \in$* NP *such that*

$$L = \begin{cases} (L_1 - L_2) \cup \cdots \cup (L_{k-2} - L_{k-1}) \cup L_k & \text{if } k \text{ is odd,} \\ (L_1 - L_2) \cup \cdots \cup (L_{k-1} - L_k) & \text{if } k \text{ is even.} \end{cases}$$

The union $\bigcup_{k\geq 1} \mathrm{NP}(k)$ *of all the levels of the boolean hierarchy is denoted by* BH.

For a set A, χ^A denotes the characteristic function of A. χ_k^A is the k-ary characteristic function of A, $\#_k^A$ gives the number out of k words, which are in A, and $\oplus_k^A$ is the parity of this number, i.e.,

$$\begin{aligned} \chi_k^A(x_1,\dots,x_k) &= \chi^A(x_1)\dots\chi^A(x_k), \\ \#_k^A(x_1,\dots,x_k) &= \sum_{i=1}^{k}\chi^A(x_i), \\ \oplus_k^A(x_1,\dots,x_k) &= \#_k^A(x_1,\dots,x_k) \bmod 2. \end{aligned}$$

The unbounded version of χ_k^A is $\chi_\omega^A = \bigcup_{k\geq 1}\chi_k^A$.

Clearly, every set $L \in \mathrm{NP}(k)$ is k-truth-table reducible to SAT, i.e., $L \in \mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ (here and in the following, the composition operator $\circ$ takes precedence over $//$). Every set that is k-truth-table reducible to SAT is in $\mathrm{NP}(k+1)$ ([28], see also [8]); thus

$$\mathrm{NP}(k) \subseteq \mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP} \subseteq \mathrm{NP}(k+1). \tag{5.1}$$

Since $\mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ is closed under complementation, the classes in (5.1) are all different unless BH (and therefore PH [20]) collapses [15]. It is interesting to note that a P machine needs only to know the parity of the number of k queries in SAT in order to decide a set in $\mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ ([43], see also [8]),

$$\mathrm{P}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP} = \mathrm{P}//\#_k^{\mathrm{SAT}} \circ \mathrm{FP} = \mathrm{P}// \oplus_k^{\mathrm{SAT}} \circ \mathrm{FP}. \tag{5.2}$$

We show in the next theorem that the first equality in (5.2) also holds for the nondeterministic counterparts of these classes, which furthermore coincide with the $(2k+1)$th level of the boolean hierarchy. Since, as it is easily seen, $\mathrm{NP}// \oplus_k^{\mathrm{SAT}} \circ\mathrm{FP}$ is contained in $\mathrm{P}//\chi_{k+2}^{\mathrm{SAT}} \circ \mathrm{FP}$, we cannot replace P by NP, for $k \geq 2$, in the second equality of (5.2), unless BH, and thus PH, collapse. It is an open question whether also the classes $\mathrm{NP}// \oplus_k^{\mathrm{SAT}} \circ\mathrm{FP}$ characterize some levels of the boolean hierarchy.

We denote the bitwise ordering on strings of the same length by $\preceq$, i.e., $a_1 \dots a_k \preceq b_1 \dots b_k$, if $a_i \leq b_i$ for $i = 1, \dots, k$.

THEOREM 5.2. $\mathrm{NP}(2k+1) = \mathrm{NP}//\#_k^{\mathrm{SAT}} \circ \mathrm{FP} = \mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ *for all* $k \geq 0$.

Proof. Let L be in $\mathrm{NP}(2k+1)$. Then there exist sets $L_1, \dots, L_{2k+1} \in \mathrm{NP}$ such that $L = (L_1 - L_2) \cup \cdots \cup (L_{2k-1} - L_{2k}) \cup L_{2k+1}$. Define the function

$$f(x) = \sum_{i=1}^{k} \chi^{L_{2i}}(x),$$

and let A be the set defined as

$$\begin{aligned} \langle x, m\rangle \in A \quad \Leftrightarrow \quad & x \in L_{2k+1} \text{ or there exist } I \subseteq \{2i \mid x \in L_{2i}\} \text{ and } j \leq k \\ & \text{such that } |I| = m,\ x \in L_{2j-1}, \text{ and } 2j \notin I. \end{aligned}$$

Clearly, $f \in \#_k^{\mathrm{SAT}} \circ \mathrm{FP}$, $A \in \mathrm{NP}$, and it holds that $x \in L$ if and only if $\langle x, f(x)\rangle \in A$. To see this, observe that there is exactly one set $I \subseteq \{2i \mid x \in L_{2i}\}$ of cardinality $f(x)$, namely, $I = \{2i \mid x \in L_{2i}\}$. Therefore, $L \in \mathrm{NP}//\#_k^{\mathrm{SAT}} \circ \mathrm{FP}$.

It is clear that $\mathrm{NP}//\#_k^{\mathrm{SAT}} \circ \mathrm{FP} \subseteq \mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$. It remains to show that $\mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP} \subseteq \mathrm{NP}(2k+1)$. For this we adapt a proof technique used by Buss and Hay [14]. Let L be in $\mathrm{NP}//\chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$, i.e., there exist a set $A \in \mathrm{NP}$ and a function $f \in \mathrm{FP}$ such that $x \in L$ if and only if $\langle x, \chi_k^{\mathrm{SAT}}(f(x))\rangle \in A$. For $m \geq 0$, consider the NP sets

$$\begin{aligned} B_m &= \{x \mid \#_k^{\mathrm{SAT}}(f(x)) \geq m\}, \\ A_m &= \left\{ x \mid \exists a = a_1 \dots a_k \in \Sigma^k : \sum_{i=1}^{k} a_i = m,\ a \preceq \chi_k^{\mathrm{SAT}}(f(x)), \text{ and } \langle x, a\rangle \in A \right\}. \end{aligned}$$

It is easy to see that $A_m \subseteq B_m$ and $B_{m+1} \subseteq B_m$. Furthermore, $B_m - B_{m+1} = \{x | \#_k^{\mathrm{SAT}}(f(x)) = m\}$ and $A_m - B_{m+1} = \{x \in B_m - B_{m+1} \mid \langle x, \chi_k^{\mathrm{SAT}}(f(x))\rangle \in A\}$. The latter equality follows from the fact that for any $x \in B_m - B_{m+1}$, there is only one string $a \in \Sigma^k$ containing m 1's and fulfilling $a \preceq \chi_k^{\mathrm{SAT}}(f(x))$, namely, $a = \chi_k^{\mathrm{SAT}}(f(x))$. Therefore, $x \in L$ if and only if $x \in A_m - B_{m+1}$, for some $m \leq k$. Since $B_{k+1} = \emptyset$, it follows that $L = (A_0 - B_1) \cup \cdots \cup (A_{k-1} - B_k) \cup A_k$. □

Hemachandra [18] (see also Buss and Hay [14]) has shown that the classes $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$ and $\mathrm{P}//\chi_\omega^{\mathrm{SAT}} \circ \mathrm{FP}$ coincide. By a slight modification in the above proof we get the following corollary, yielding a further characterization of $\mathrm{P}^{\mathrm{NP}[O(\log n)]}$.

COROLLARY 5.3. $\mathrm{P}//\chi_\omega^{\mathrm{SAT}} \circ \mathrm{FP} = \mathrm{NP}//\chi_\omega^{\mathrm{SAT}} \circ \mathrm{FP}$.

Beigel [8] shows that $\mathrm{P}//\mathrm{OptP}[k] = \mathrm{P}//\chi_{2^k-1}^{\mathrm{SAT}} \circ \mathrm{FP}$. From Theorem 5.2 and the following Theorem 5.4, it follows that this equation remains valid when P is replaced by NP. Theorem 5.4 restates an observation in [26] that $\#_{2^k-1}^{\mathrm{SAT}}$ is complete for $\mathrm{OptP}[k]$.

THEOREM 5.4 [26]. $\mathrm{OptP}[k] = \#_{2^k-1}^{\mathrm{SAT}} \circ \mathrm{FP} \cup \#_{2^k-1}^{\overline{\mathrm{SAT}}} \circ \mathrm{FP}$ *for all* $k \geq 0$.

COROLLARY 5.5. $\mathrm{NP}(2^{k+1} - 1) = \mathrm{NP}//\mathrm{OptP}[k]$ *for all* $k \geq 0$.

Ladner, Lynch, and Selman [30] transformed the recursion-theoretic truth-table reducibility into complexity theory. They also give a definition of a nondeterministic truth-table reduction in the following way: A is nondeterministically truth-table reducible to B if there exists an NP transducer G (the *generator*) and an NP machine E (the *evaluator*) such that for every x,

$$\begin{aligned} x \in A \Leftrightarrow {} & \text{there exists a branch of } G(x) \text{ yielding an output} \\ & y = \langle y_1, \ldots, y_k \rangle \text{ such that } E(x, \chi_\omega^B(y_1, \ldots, y_k)) \text{ accepts.} \end{aligned}$$

It is known that this definition is equivalent with the nondeterministic Turing reducibility [30] and therefore does not lead to a new reducibility notion. We modify the above definition by restricting the generator G to be a *single-valued* NP transducer, i.e., the output must be the same on every accepting branch. Let NPSV be the set of functions computed by single-valued NP transducers [11].

This reducibility first appeared in [11] (there denoted by NP.UNIF.ALL) and was explicitly called *nondeterministic truth-table reducibility* by Book and Ko [10]. Subsequently, Book and Tang [12] and Rich [35] introduced the following terminology.

DEFINITION 5.6. *A set A is* nondeterministically truth-table reducible *to B ($A \leq_{\mathrm{tt}}^{\mathrm{NP}} B$), if $A \in \mathrm{NP}//\chi_\omega^B \circ \mathrm{NPSV}$. A is* nondeterministically k-truth-table reducible *to B ($A \leq_{k\text{-tt}}^{\mathrm{NP}} B$), if $A \in \mathrm{NP}//\chi_k^B \circ \mathrm{NPSV}$. For a class $\mathcal{C}$ of sets let $\mathrm{NP}_{\mathrm{tt}}^{\mathcal{C}}$ be the class $\{A \mid \exists\, B \in \mathcal{C} : A \leq_{\mathrm{tt}}^{\mathrm{NP}} B\}$ of all sets $\leq_{\mathrm{tt}}^{\mathrm{NP}}$-reducible to some set in $\mathcal{C}$, and let $\mathrm{NP}_{k\text{-tt}}^{\mathcal{C}} = \{A \mid \exists\, B \in \mathcal{C} : A \leq_{k\text{-tt}}^{\mathrm{NP}} B\}$.*

In [11], it is shown that there exist recursive sets A and B such that $A \leq_T^{\mathrm{NP}} B$ and $A \not\leq_{\mathrm{tt}}^{\mathrm{NP}} B$. This means that $\leq_{\mathrm{tt}}^{\mathrm{NP}}$ is properly stronger than $\leq_T^{\mathrm{NP}}$. The question whether $\leq_{\mathrm{tt}}^{\mathrm{P}}$ is properly stronger than $\leq_{\mathrm{tt}}^{\mathrm{NP}}$ is equivalent to the P =?NP problem [11], [35]. However, as we will see in Corollary 5.8, every set A that is nondeterministically truth-table reducible to some NP complete set B is also deterministically truth-table reducible to B, i.e., B is NP complete and

$$A \leq_{\mathrm{tt}}^{\mathrm{NP}} B \Rightarrow A \leq_{\mathrm{tt}}^{\mathrm{P}} B.$$

Thus, we have the surprising result that while the definition in [30] of a nondeterministic truth-table reduction was too weak, the definition of Rich seems to be too strong to yield a new reduction class between $\{L \mid L \leq_{\mathrm{tt}}^{\mathrm{P}} \mathrm{SAT}\}$ and $\{L \mid L \leq_T^{\mathrm{NP}} \mathrm{SAT}\}$. As a further consequence of Theorem 5.7, we get a characterization of the odd levels of the boolean hierarchy in terms of the nondeterministic k-truth-table reducibility notion.

THEOREM 5.7. (i) $\chi_k^{\mathrm{SAT}} \circ \mathrm{NPSV} = \chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$ *for all $k \geq 1$,*
(ii) $\chi_\omega^{\mathrm{SAT}} \circ \mathrm{NPSV} \subseteq \mathrm{FP}_{\mathrm{tt}}^{\mathrm{SAT}}$.

Proof. To see (i) let f be in NPSV and define the NP set

$$A = \{\langle x, m \rangle \mid \exists z_1, \ldots, z_k : f(x) = (z_1, \ldots, z_k) \text{ and } z_m \in \mathrm{SAT}\}.$$

Then $\chi_k^{\mathrm{SAT}}(f(x)) = \chi_k^A(\langle x, 1 \rangle, \ldots, \langle x, k \rangle)$ for all x, and thus, $\chi_k^{\mathrm{SAT}} \circ f \in \chi_k^A \circ \mathrm{FP} \subseteq \chi_k^{\mathrm{SAT}} \circ \mathrm{FP}$.

For the proof of (ii) define the NP set

$$B = \{\langle x, k, m, b \rangle \mid \exists z_1, \ldots, z_k : f(x) = (z_1, \ldots, z_k) \text{ and } b \leq \chi^{\mathrm{SAT}}(z_m)\}$$

and observe that $\chi_\omega^{\mathrm{SAT}}(f(x))$ can be read off B's answers to the parallel queries $\langle x, k, m, b \rangle$, for $k = 1, \ldots, p(|x|)$, $m = 1, \ldots, k$, and $b = 0, 1$, where p is a polynomial bounding the running time of the NP transducer that computes f. □

COROLLARY 5.8. (i) $\mathrm{NP}(2k + 1) = \mathrm{NP}_{k\text{-tt}}^{\mathrm{NP}}$, *for all $k \geq 1$,*
(ii) $\mathrm{P}_{\mathrm{tt}}^{\mathrm{NP}} = \mathrm{NP}_{\mathrm{tt}}^{\mathrm{NP}}$.

Remark 5.9. Book and Tang [12] especially consider the $O(\log n)$ bounded version $\leq_{\log n\text{-tt}}^{\mathrm{NP}}$ of the nondeterministic truth-table reduction obtained by logarithmically bounding the number of queries produced by the NPSV generator. It follows from (appropriately modified versions of) Theorem 5.7, Corollary 5.3, and Lemma 3.6 that

$$\mathrm{NP}^{\mathrm{NP}}_{\log n\text{-tt}} = \mathrm{P}^{\mathrm{NP}}_{\log n\text{-tt}} = \mathrm{P}^{\mathrm{NP}[O(1)+\log\log n]}.$$

This class is also considered by Wagner [42] (there denoted by $\mathrm{P}^{\mathrm{NP}}_{||}[O(\log n)]$), who shows that it coincides with the class of languages that are full-truth-table reducible[2] to SAT. As a consequence, it follows that $A \leq^{\mathrm{NP}}_{\log n\text{-tt}}$ SAT if and only if A is full-truth-table reducible to SAT.

Remark 5.10. Book and Tang [12] generalized the nondeterministic truth-table reducibility to a Σ_k truth-table reducibility by giving the generator and the evaluator access to a Σ_{k-1} oracle: A is Σ_k truth-table reducible to B ($A \leq^{\Sigma_k}_{\mathrm{tt}} B$) if $A \in \Sigma_k // \chi^B_\omega \circ \mathrm{NPSV}^{\Sigma_{k-1}}$. For a class $\mathcal{C}$ of sets let $\Sigma^{\mathcal{C}}_{k,\mathrm{tt}}$ be the class $\{A \mid \exists B \in \mathcal{C} : A \leq^{\Sigma_k}_{\mathrm{tt}} B\}$. Then Corollary 5.8(ii) generalizes to

$$\Sigma^{\Sigma_k}_{k,\mathrm{tt}} = \mathrm{P}^{\Sigma_k}_{\mathrm{tt}} = \mathrm{P}^{\Sigma_k[O(\log n)]},$$

i.e., every set that is Σ_k truth-table reducible to a set in Σ_k is already (deterministically) truth-table reducible to a set in Σ_k.

Thierauf [38] showed that allowing the generator in the nondeterministic truth-table reduction to produce polynomially many different outputs (i.e., to compute an NPPV function [11]) does not increase the class of sets reducible to SAT.

THEOREM 5.11 [38]. *Let L be a set, G an* NPPV *transducer, and E an* NP *set such that*

$$x \in L \Leftrightarrow \exists \langle y_1, \ldots, y_k\rangle \in \mathrm{out}_G(x) : \ \langle x, \chi^{\mathrm{SAT}}_\omega(y_1, \ldots, y_k)\rangle \in E.$$

Then L is in $\mathrm{P}^{\mathrm{NP}}_{\mathrm{tt}}$.

We end this section by proving that also the strong nondeterministic truth-table reducibility, introduced by Long [31], when applied to SAT, is only as powerful as $\leq^{\mathrm{P}}_{\mathrm{tt}}$. As in the definition of Ladner, Lynch, and Selman [30], the generator in a strong nondeterministic truth-table reduction can produce exponentially many different outputs, but the evaluator either has to accept all the outputs or it has to reject all of them.

DEFINITION 5.12 [31]. *A is* strongly nondeterministically truth-table reducible *to B ($A \leq^{\mathrm{SN}}_{\mathrm{tt}} B$) if there is an* NP *transducer G and a* P *machine E such that for all x the set $\mathrm{out}_G(x)$ is nonempty, and for all $\langle y_1, \ldots, y_k\rangle$ in $\mathrm{out}_G(x)$, $E(x, \chi^B_\omega(y_1, \ldots, y_k)) = \chi^A(x)$. For a class of sets $\mathcal{C}$ we denote by* $\mathrm{SN}^{\mathcal{C}}_{\mathrm{tt}}$ *the class $\{A \mid \exists B \in \mathcal{C} : A \leq^{\mathrm{SN}}_{\mathrm{tt}} B\}$.*

Clearly, $\leq^{\mathrm{SN}}_{\mathrm{tt}}$ lies in strength between $\leq^{\mathrm{P}}_{\mathrm{tt}}$ and $\leq^{\mathrm{NP}}_{T}$. Long [31] showed that $\leq^{\mathrm{SN}}_{\mathrm{tt}}$ is properly stronger than $\leq^{\mathrm{NP}}_{T}$ by constructing two sets A and B such that $A \not\leq^{\mathrm{SN}}_{\mathrm{tt}} B$ and $A \leq^{\mathrm{NP}}_{T} B$. The question whether $\leq^{\mathrm{P}}_{\mathrm{tt}}$ is properly stronger than $\leq^{\mathrm{SN}}_{\mathrm{tt}}$ is closely related to two major open questions in complexity theory [31]:

$$\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP} \ \Rightarrow\ \leq^{\mathrm{P}}_{\mathrm{tt}} \neq \leq^{\mathrm{SN}}_{\mathrm{tt}} \ \Rightarrow\ \mathrm{P} \neq \mathrm{NP}.$$

THEOREM 5.13. $\mathrm{SN}^{\mathrm{NP}}_{\mathrm{tt}} = \mathrm{P}^{\mathrm{NP}}_{\mathrm{tt}}$.

Proof. Let L be in $\mathrm{SN}^{\mathrm{NP}}_{\mathrm{tt}}$ via a generator G, an evaluator E, and a set $A \in$ NP. In order to decide membership of a given input x, it suffices to find out whether there is some output $\langle y_1, \ldots, y_k\rangle$ of $G(x)$ such that E accepts $\langle x, \chi^A_\omega(y_1, \ldots, y_k)\rangle$. But this becomes an NP problem, provided that the maximum number $\#^A_\omega(y_1, \ldots, y_k)$ of yes-answers from A over all outputs $\langle y_1, \ldots, y_k\rangle$ of $G(x)$ is given along with the input x.

[2] A set A is full-truth-table reducible [28], [14] to a set B if there is a function $g \in$ FP such that for all x, $g(x)$ is of the form $\langle a_0 \ldots a_{2^m-1}, y_1, \ldots, y_m\rangle$, where $a_i \in \{0, 1\}$ $(0 \leq i \leq 2^m - 1)$ and $y_i \in \Sigma^*$ $(1 \leq i \leq m)$, and it holds that $x \in A \Leftrightarrow a_j = 1$, where j is the number whose binary representation is given by $\chi^B_\omega(y_1, \ldots, y_m)$.

More precisely, define the function

$$h(x) = \max\{\#^A_\omega(y_1, \ldots, y_k) \mid \langle y_1, \ldots, y_k\rangle \in \mathrm{out}_G(x)\},$$

and let B be the set defined as

$$\langle x, m\rangle \in B \quad \Leftrightarrow \quad \exists a = a_1 \ldots a_k \in \Sigma^k \ \exists \langle y_1, \ldots, y_k\rangle \in \mathrm{out}_G(x) : \\ \sum_{i=1}^{k} a_i = m, \quad a \preceq \chi^A_\omega(y_1, \ldots, y_k) \text{ and } E(x, a) = 1.$$

Then $h \in \mathrm{OptP}[O(\log n)]$ and $B \in \mathrm{NP}$, and it holds for all x that $x \in L$ if and only if $\langle x, h(x)\rangle \in B$, i.e., L is in $\mathrm{NP}//\mathrm{OptP}[O(\log n)] = \mathrm{P}//\mathrm{OptP}[O(\log n)]$. □

Note that by the above proof, Theorem 5.13 remains true if the evaluator E is allowed to be an NP machine.

Acknowledgments. For helpful conversations and suggestions on this work we are grateful to V. Arvind, L. Hemachandra, and U. Schöning. We also thank the two referees for their valuable suggestions.

REFERENCES

[1] A. Amir and W. Gasarch, *Polynomial terse sets*, Proceedings 2nd Structure in Complexity Theory Conference, IEEE Computer Society, 1987, pp. 259-269.

[2] T. Baker, J. Gill, and R. Solovary, *Relativizations of the* P =?NP *question*, SIAM J. Comput., 4 (1975), pp. 431–442.

[3] J. Balcázar, *Nondeterministic witnesses and nonuniform advice*, Proc. 4th Structure in Complexity Theory Conference, IEEE Computer Society, 1989, pp. 259–269.

[4] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity I*, Springer, New York, 1987.

[5] J. Balcázar and U. Schöning, *Logarithmic advice classes*, Theoret. Comput. Sci., 99 (1992), pp. 279–290.

[6] R. Beigel, *Query-limited reducibilities*, Dissertation, Stanford University, 1987.

[7] ———, *Perceptrons, PP, and the polynomial hierarchy*, Proc. 7th Structure in Complexity Theory Conference, IEEE Computer Society, 1989, pp. 14–19.

[8] ———, *Bounded queries to* SAT *and the Boolean hierarchy*, Theoret. Comput. Sci., 84 (1991), pp. 199–223.

[9] R. Beigel, L. Hemachandra, and G. Wechsung, *Probabilistic polynomial time is closed under parity reductions*, Inform. Process. Lett., 37 (1991), pp. 91–94.

[10] R. Book, and K. Ko, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[11] R. Book, T. Long, and A. Selman, *Quantitative relativizations of complexity classes*, SIAM J. Comput., 13 (1984), pp. 461–487.

[12] R. Book and S. Tang, *Characterizing polynomial complexity classes by reducibilities*, Math. Systems Theory, 23 (1990), pp. 165–174.

[13] ———, *A note on sparse sets and the polynomial-time hierarchy*, Inform. Process. Lett., 33 (1989), pp. 141–143.

[14] S. Buss and L. Hay, *On truth-table reducibility to* SAT, Inform. and Comput., 91 (1991), pp. 86–102.

[15] J. Cai, T. Gundermann, J. Hartmanis, L. Hemachandra, V. Sewelson, K. Wagner, and G. Wechsung, *The boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

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

[17] J. Hartmanis and Y. Yesha, *Computation times of* NP *sets of different densities*, Automata, Languages and Programming, Lecture Notes in Comput. Sci. 154, Springer-Verlag, New York, 1983, pp. 319–330.

[18] L. Hemachandra, *The strong exponential hierarchy collapses*, Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 110–122.

[19] L. Hemachandra, Personal communication, Barcelona 1990.

[20] J. Kadin, *The polynomial time hierarchy collapses if the boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.

[21] ———, P^{NP} *and sparse Turing-complete sets for* NP, J. Comput. System Sci., 39 (1989), pp. 282–298.

[22] J. Kämper, *Non-uniform proof systems: A new framework to describe non-uniform and probabilistic complexity classes*, Theoret. Comput. Sci., 85 (1991), pp. 305–331.

[23] R. KARP AND R. LIPTON, *Turing machines that take advice*, Enseign. Math., 28 (1982), pp. 191–209.

[24] K. KO, *Relativized polynomial time hierarchies having exactly k levels*, SIAM J. Comput., 18 (1989), pp. 392–408.

[25] J. KÖBLER, *Strukturelle Komplexität von Anzahlproblemen*, Ph.D. Thesis, University of Stuttgart, 1989.

[26] J. KÖBLER, U. SCHÖNING, AND J. TORÁN, *On counting and approximation*, Acta Inform., 26 (1989). pp. 363–379.

[27] J. KÖBLER, U. SCHÖNING, S. TODA, AND J. TORÁN, *Turing machines with few accepting computations and low sets for* PP, J. Comput. System Sci., 44 (1992), pp. 272–286.

[28] J. KÖBLER, U. SCHÖNING, AND K. WAGNER, *The difference and truth-table hierarchies of* NP., Theoret. Inform. Appl., 21 (1987), pp. 419–435.

[29] M. KRENTEL, *The complexity of optimization problems*, J. Comput. System Sci., 36 (1988), pp. 490–509.

[30] R. LADNER, N. LYNCH, AND A. SELMAN, *A comparison of polynomial-time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.

[31] T. LONG, *Strong nondeterministic polynomial-time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.

[32] ———, *On restricting the size of oracles compared with restricting access to the oracle*, SIAM J. Comput., 14 (1985), pp. 585–597.

[33] S. MAHANEY, *Sparse complete sets for* NP : *solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[34] N. PIPPENGER, *On simultaneous resource bounds*, Proc. 20th Symposium on Foundations of Computer Science, IEEE Computer Society, 1979, pp. 307–311.

[35] C. RICH, *Positive relatvizations of the* P =?NP *problem*, J. Comput. System Sci., 38 (1989), pp. 511–523.

[36] U. SCHÖNING, *Complexity and structure*, Lecture Notes in Comput. Sci. 211, Springer-Verlag, New York, 1986.

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

[38] T. THIERAUF, NP-*uniforme Komplexitätsklassen*, Ph.D. Thesis, University of Ulm, 1990.

[39] S. TODA, *PP is as hard as the polynomial-time hierarchy*, SIAM J. Comput., 20 (1991), pp. 865–877.

[40] ———, *Restricted relativizations of probabilistic polynomial time*, Theoret. Comput. Sci., 93 (1992), pp. 265–277.

[41] K. WAGNER, *On restricting the access to an* NP-*oracle*, Automata, Languages and Programming, Lecture Notes in Comput. Sci. 317, Springer-Verlag, New York, 1988.

[42] ———, *Bounded query computations*, SIAM J. Comput., 19 (1990), pp. 833–846.

[43] G. WECHSUNG, *On the boolean closure of NP*, Proc. 5th Conference on Fundamentals of Computation Theory, Lecture Notes in Comput. Sci. 199, Springer-Verlag, New York, 1985, pp. 485–493. (An unpublished precursor of this paper was coauthored by K. Wagner.)

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 276–292, April 1994

DETERMINISTIC SIMULATIONS OF PRAMS ON BOUNDED DEGREE NETWORKS*

KIERAN T. HERLEY†‡ AND GIANFRANCO BILARDI†§

Abstract. The problem of simulating a PRAM with n processors and memory size $m \geq n$ on an n-node bounded degree network is considered. A deterministic algorithm is presented that simulates an arbitrary PRAM step in $O((\log n \log m)/\log\log n)$ time in the worst case on an expander-based network. By extending a previously established lower bound, it is shown that the proposed simulation is optimal whenever $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^{\alpha}})$ for some positive real constants ϵ and α.

Key words. parallel computation, shared memory machines, simulations, networks of processors, expander graphs

AMS subject classification. 68Q05

1. Introduction.

Parallel random access machines (PRAMs) play a central role in the study of parallel computation and in the development of parallel algorithms [KR90]. A PRAM is essentially a set of synchronous processors connected to a shared memory. The basic feature of the PRAM is that references to distinct memory cells can be made simultaneously by different processors. A number of different variants of the PRAM model have been proposed [FW78], [Gol78], [Sch80], [LPV81], [SV81], [Kuc82], [Sni85], [BH85] that allow simultaneous references to the same memory cell in varying degrees. By ignoring several of the constraints that arise in the physical realization of a multiprocessor, the PRAM model provides an attractive framework for algorithm design. However, the running time of an algorithm on a PRAM may be a poor estimate of its performance on a realistic multiprocessor [Sny86].

Two important constraints not captured by PRAMs but shared by most realistic parallel machines are as follows: (i) the granularity of the memory (the memory is partitioned into banks or modules, each of which can respond to only one request per machine cycle), and (ii) the fact that the individual components of the machine (processors and memory modules) can be connected directly only to a small number of other components. The Bounded degree network (BDN) of processors and memory modules captures both constraints and hence is a more realistic model of parallel computers than the PRAM. A BDN consists of a synchronous collection of nodes, each equipped with a processor and a private local memory. Each node is connected to a constant number (independent of machine size) of other nodes by means of bidirectional communication channels. It is natural to wonder how well a PRAM computation can be simulated by a suitably chosen BDN. This question has received considerable attention in the literature, and we briefly review some of the results here.

Several researchers have observed that simulating a PRAM whose memory size m is linear in the number of processors n is essentially a routing problem (see [LPV81], [BH85], for example). The idea is that each node in the simulating network holds $O(1)$ PRAM memory cells; accessing a cell involves communicating with the node that holds it. In this paper we focus on the simulation of PRAMs where m, the memory size, is significantly larger than n, the number of PRAM processors. We will refer to such a PRAM as an (n, m)-PRAM. In this

*Received by the editors February 22, 1989; accepted for publication (in revised form) October 20, 1992.

†Computer Science Department, Cornell University, Ithaca, New York 14853.

‡Current address, Department of Computer Science, University College, Cork, Ireland. The work of this author was supported in part by the National Science Foundation grant DCR-86-02307 and by the Joint Services Electronics Program contract F49620-87-C-0044.

§Current address, Dipartimento di Elettronica ed Informatica, Universitá di Padova, Via Gradenigo 6/A, 35131 Padova, Italy. The work of this author was supported in part by the National Science Foundation grant MIP-86-02256 and by the Joint Services Electronics Program contract F49620-87-C-0044.

case the chief obstacle to efficient simulation is *congestion*. Since only a single value can be accessed from any particular memory module in one BDN machine cycle, the simulation algorithm must limit the amount of information accessed from each node.

One approach, proposed in [MV84] and further developed subsequently in a series of papers [Upf84], [LPP88], [KU88], [LMR88], [Val88], [Ran91], distributes the address space of the PRAM among the modules of the BDN according to some suitably chosen hash function. The objective is to ensure that, with high probability, the congestion at any node of the network is low, and hence performance is good. In particular [Ran91] proposes a protocol for an n-node butterfly network that simulates T steps of an (n, m)-PRAM in $O(T \log n)$ time, with high probability, assuming that m is polynomial in n. The protocol is deterministic, except for the random choice of the hash function that defines the address map.

Another approach, also proposed in [MV84], relies on representing the contents of a PRAM memory cell by using several BDN cells. This approach forms the basis for a number of deterministic simulation schemes [UW87], [AHMP87], [LPP90] including the one proposed in this paper. The goal of these schemes is to provide good *worst-case* performance.

In [UW87], a scheme is advanced to simulate an arbitrary step of an (n, m)-PRAM $(m \geq n)$ by an n-node BDN in time $O(\log n \log m \log\log m)$. Nodes of the BDN consist of a RAM processor and a memory module. They are interconnected as in the graph (V, E_{AKSL}), Leighton's modification of the AKS sorting network [AKS83], [Lei85]. This time bound was subsequently improved to $O(\log n \log m)$ in [AHMP87] and is further improved to $O(\log n \log m / \log\log n)$ in the present paper. Our network is also an n-node BDN, $(V, E_{\text{exp}} \cup E_{\text{AKSL}} \cup E_{\text{tree}})$ where (V, E_{exp}) is a graph based on square-root expanders [PU89], [LPS86], (see also [LPS88]) and (V, E_{tree}) is a complete binary tree. A simulation with a running time of $O(\log^2 n/\log\log n + \log(m/n)\log n)$ has been obtained in [HMP89], which matches the performance of our scheme only for $m = n^{1+O(1/\log\log n)}$.

An $\Omega((\log n \log m)/\log\log m)$ lower bound for the time to simulate a step of an (n, m)-PRAM by an n-node BDN was established independently in [AHMP87] and [KU88] assuming that, for some positive constant ϵ, $\Omega(n^{2+\epsilon}) \leq m \leq 2^{O(\sqrt{n \log n})}$. By refining the proof technique of [AHMP87] and [KU88], we obtain a lower bound of $\Omega((\log(m/n))^2/\log\log(m/n))$ for $m \leq O(n^{2+\epsilon})$. A similar result has been obtained in [HMP89]. The performance of the simulation described in this paper matches the cited lower bound as long as $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^\alpha})$, for some positive real constants ϵ and α.

The lower bounds just mentioned are based on the so-called point-to-point assumption, technically defined in §5. Essentially, it is assumed that when storing a new value of a PRAM variable in different memory locations, a BDN processor needs to issue a distinct message for each location. There is really no reason why a simulation should be restricted in this way. Nevertheless, the lower bound is interesting for the following reasons. All the simulations cited above are essentially point to point. At least for the range of m where the simulations of this paper are optimal, the lower bound indicates that the only hope for an improvement lies in "combining" techniques for messages from the same source carrying the same information. Moreover, even with the point-to-point limitation, the lower bound arguments are not straightforward, and they provide valuable insights on the problem. A recent study reports an $\Omega(\log m)$ lower bound that does not rely on the point-to-point assumption [HMP89].

In [LPP90] a PRAM simulation is presented on a BDN consisting of an $n \times n$ mesh-of-trees [Lei84], [NMB83], where the n roots of the trees are RAM processors, while the other $\Theta(n^2)$ nodes are packet-switching elements. The scheme is attractive for the regularity and simplicity of the interconnection. For m polynomial in n, a PRAM step is simulated in time $O(\log^2 n/\log\log n)$, the same as that achieved by our simulation for this range of m.

However, it should be noted that the use of $\Theta(n^2)$ nodes, albeit mere switches, makes the cited lower bound inapplicable to the mesh-of-trees.

Even under the point-to-point assumption, optimal simulations are not known for small memory ($m < O(n^{1+\epsilon})$), or for very large memory ($m > \Omega(2^{(\log n)^{\alpha}})$). For small memory, simulations faster than those given in this paper have been proposed in [Her89].

Deterministic PRAM simulations on BDNs typically lead to a BDN memory size considerably larger than m, the PRAM memory size. There are two reasons for this phenomenon: (a) several copies of each PRAM variable are kept in the BDN, and (b) the memory map indicating the location of such copies needs to be stored. Simulations using only $O(1)$ copies per variable have been proposed in [HP89] for a mesh-of-trees interconnection. Memory maps with substantially more compact representations than those previously considered are discussed in [Her90].

In most of the studies referred to above, the number n of nodes in the BDN is assumed to be proportional to the number of processors in the PRAM being simulated and is independent of the size m of the PRAM memory. It is certainly of interest to investigate simulations where the size of the BDN is allowed to depend on both n and m. In this direction it is worth mentioning the scheme of [Vis84], which has running time $O(\log n + \log m)$ on a BDN of $O(n \log n + m \log m)$ nodes.

The remainder of this paper is organized as follows. Section 2 is devoted to the formulation of the problem. Section 3 discusses the memory organization, i.e., the method by which the memory of the PRAM is represented by the memory of the BDN. Introducing the graph-theoretic notion of generalized expander, we investigate the trade-off between the number of copies and memory congestion in organizations of the type proposed in [UW87]. The choice of an appropriate point in this trade-off is critical in order to balance the work needed to extract the relevant information from the memory modules with the work needed to route this information to the intended processors. Section 4 describes and analyzes the memory access algorithm. Section 5 is devoted to lower bounds. Section 6 concludes the paper by indicating a number of interesting open problems.

2. Problem formulation. We consider an (n, m)-PRAM with processors, P_1^{PRAM}, P_2^{PRAM}, …, P_n^{PRAM} sharing a set U of m memory cells, which will also be referred to as *variables*. Our goal is to simulate such a PRAM by a BDN of n nodes (P_1, M_1), (P_2, M_2), …, (P_n, M_n) where P_i is a RAM processor and M_i is a memory module. We are interested in simulation algorithms that are deterministic and on line, i.e., the simulation of a step is independent of future steps.

We assume that the instruction to be executed by PRAM processor P_i^{PRAM} at a given step is available to the corresponding BDN processor P_i at the beginning of the simulation of this step. Since simulation of arithmetic and logical operations is straightforward, we can regard a generic PRAM step simply as an n-tuple of memory accesses. A memory access can either be of the form read(u), $(u \in U)$, which results in the contents of the memory cell u being copied into a distinguished register of the executing processor, or of the form write(u, x), ($u \in U$, x an integer value), which results in the integer value x being written to memory cell u. For simplicity we assume that all the accesses in a given step are of the same t;pe (read or write).

In any simulation scheme, upon termination of the simulation of a given PRAM step, the state of the PRAM memory must be uniquely reconstructible from the state of the BDN memory. The method chosen for representing the state of the PRAM memory by the state of the BDN memory is referred to as the *memory organization*. The algorithm that probes the state of the BDN memory when simulating a read step and that modifies it when simulating a write step is called the *memory access algorithm*.

A simulation scheme is *consistent* if, after the simulation of an arbitrary sequence of PRAM steps, the simulation of a step where processor P_i^{PRAM} executes read(u) results in loading the distinguished register of P_i with the correct value of u. The latter is of course the value assigned to u by the latest instruction of the form write(u, x).

In the next two sections we describe and analyze a new simulation scheme. Section 3 is devoted to the memory organization, and §4 to the memory access algorithm.

3. Memory organization. The idea of replicating data to enhance its accessibility is widely used in the field of distributed computing [Gif79], [Tho79]. Upfal and Wigderson [UW87] proposed a memory organization where, for each PRAM variable $u \in U$, the BDN maintains $2c - 1$ copies in the memory modules of a set $\Gamma(u) = \{\gamma_1(u), \gamma_2(u), \ldots, \gamma_{2c-1}(u)\}$ of distinct nodes. Their scheme adheres to the following read/write discipline. Each copy contains a value and a timestamp indicating the time at which that copy was last written. When simulating a step where u is written, at least c copies of u are updated. When simulating a step where u is read, at least c copies of u are retrieved: the one with the most recent timestamp is guaranteed to contain the correct value of u. We will adopt the same read/write discipline.

In the memory organizations employed in our simulation the location of the copies of a particular variable does not vary with time. This type of memory organization can be modelled conveniently by a bipartite graph $G = (U, V, E)$ where U is the set of PRAM variables, V is the set of the nodes of the BDN, and an edge (u, v) represents the fact that the node v contains a copy of the variable u.

Let $S \subseteq U$ and let $F \subseteq E$ have exactly k edges incident upon each $u \in S$. We call F a *k-bundle* for S. We say that the subset of V on which the edges of F are incident is a *k-neighborhood* of S and denote it by $\Gamma_F(S)$. When the read/write discipline scheme described above is adopted, the simulation of a PRAM step accessing the variables of S requires that the BDN access all the copies of some c-bundle, say F. Clearly, this takes at least $c|S|/|\Gamma_F(S)|$ time, since at least $c|S|$ copies have to be accessed from $|\Gamma_F(S)|$ modules. It is desirable to have a memory organization with large c-neighborhoods. The preceding remarks motivate the following definition.

DEFINITION 1. *A bipartite graph $G = (U, V, E)$ with $|U| = m$ and $|V| = n$, and with each node in U having degree d is a (λ, d, c, σ)-generalized expander if, for every $S \subseteq U$ such that $|S| \leq \sigma n$ and for every c-bundle F of S, $|\Gamma_F(S)| \geq \lambda c|S|$.*

Clearly, $c \leq d \leq n$, $\lambda \leq 1$, and $\sigma \leq 1/\lambda c$ are necessary conditions for the existence of a (λ, d, c, σ)-generalized expander. The next theorem establishes the existence of certain types of generalized expanders. (The proof is based on standard types of counting arguments and can be skipped without loss of continuity.)

THEOREM 1. *Let $g_0(\lambda) \triangleq (4e^{1+\lambda})^{2/(1-\lambda)}$. For every n, m, λ, g, and c with $m \geq n$, $\lambda < 1$, $2c - 1 \leq n$, $g_0(\lambda) < g \leq 3\lambda g_0(\lambda)m/n$, and $c \geq (2/(1 - \lambda)) \log(3\lambda m/n)/ \log(g/g_0(\lambda))$, there exists a $(\lambda, 2c - 1, c, 1/\lambda cg)$-generalized expander $G = (U, V, E)$ with $|U| = m$ and $|V| = n$.*

Proof. Let $B_{m,n,2c-1}$ be the class of all bipartite graphs $G = (U, V, E)$ with *labeled* nodes and *unlabeled* edges such that $|U| = m$, $|V| = n$, and each node of U has degree $2c - 1$. Note that $|B_{m,n,2c-1}| = \binom{n}{2c-1}^m$. Let η be the fraction of graphs in $B_{m,n,2c-1}$ that are *not* $(\lambda, 2c - 1, c, 1/\lambda cg)$-generalized expanders. We will show that $\eta < 1/2$.

We begin by observing that, if $G \in B_{m,n,2c-1}$ is not a $(\lambda, 2c - 1, c, 1/\lambda cg)$-generalized expander, then there exists a triple (S, W, F) with the following properties: (i) $S \subseteq U$, $|S| \leq n/\lambda cg$; (ii) $W \subseteq V$, $|W| = \lceil \lambda c|S| - 1 \rceil$; (iii) $F \subseteq E$ is a c-bundle of S such that $\Gamma_F(S) \subseteq W$. We shall call such a triple a *witness*. If we let f_s be the fraction of all graphs in $B_{m,n,2c-1}$ having a witness (S, W, F) with $|S| = s$, then we can bound η as follows:

$$\eta \leq \sum_{1 \leq s \leq n/\lambda cg} f_s. \tag{1}$$

We now claim that

$$f_s < \binom{m}{n}\binom{n}{\lceil \lambda cs - 1\rceil}\binom{2c-1}{c}^s \left\{ \frac{\binom{\lceil \lambda cs-1\rceil}{c}}{\binom{n}{c}} \right\}^s . \tag{2}$$

Standard combinatorial arguments show that there are $\binom{m}{s}$ ways of choosing S, $\binom{n}{\lceil \lambda cs-1\rceil}$ ways of choosing W,

$$\binom{\lceil \lambda cs - 1\rceil}{c}^s$$

ways of choosing F, and a fraction

$$\left\{ \binom{2c-1}{c} \Big/ \binom{n}{c} \right\}^s$$

of the graphs in $B_{m,n,2c-1}$ for which triple (S, W, F) is a witness. Then, simple arithmetic yields (2). Use of the inequalities $\binom{m}{s} < (em/s)^s$, $\binom{n}{\lceil \lambda cs-1\rceil} < (en/\lambda cs)^{\lambda cs}$, $\binom{2c-1}{c} < 2^{2c}$, and $\binom{\lceil \lambda cs-1\rceil}{c}/\binom{n}{c} < (\lambda cs/n)^c$ in (2) yields (after some manipulations)

$$f_s < \left\{ \frac{ecm}{n} \left[4e^{\lambda} \lambda^{(1-\lambda)} (cs/n)^{1-\lambda-1/c} \right]^c \right\}^s .$$

Making use of the bounds $ec \le e^c$, and $cs/n \le 1/\lambda g$ (from $s \le n/\lambda cg$), we have

$$f_s < \left\{ \frac{\lambda m}{n} \left[4e^{1+\lambda} (1/g)^{1-\lambda-1/c} \right]^c \right\}^s .$$

From $c \ge (2/(1-\lambda)) \log(3\lambda m/n)/\log(g/g_0(\lambda))$, and $g \le 3\lambda g_0(\lambda) m/n$, we have $c \ge 2/(1-\lambda)$, and hence $1 - \lambda - 1/c \ge (1-\lambda)/2$. Since $g > g_0(\lambda) > 1$, we can reformulate the expression for f_s as

$$f_s < \left\{ \frac{\lambda m}{n} \left[4e^{1+\lambda} (1/g)^{(1-\lambda)/2} \right]^c \right\}^s$$

which can be rewritten as

$$f_s < \left\{ \frac{\lambda m}{n} \left[\frac{g_0(\lambda)}{g} \right]^{(1-\lambda)c/2} \right\}^s ,$$

where $g_0(\lambda) = (4e^{1+\lambda})^{2/(1-\lambda)}$. If $c \ge (2/(1-\lambda)) \log(3\lambda m/n)/\log(g/g_0)$, then $f_s < 1/3^s$, and, from (1), $\eta < \frac{1}{2}$. □

Upfal and Wigderson [UW87] established the existence of a family of $(\lambda, 2c-1, c, 1/(2c-1))$-generalized expanders for every $\lambda < 1/(8e^4)$, and $c \ge \beta \log m$, for some suitable constant $\beta > 0$. These expanders and similar ones define the memory organizations used in [UW87], [AHMP87], [LPP90].

Theorem 1 demonstrates that $c = \lceil (2/(1-\lambda)) \log(3\lambda m/n)/\log(g/g_0) \rceil$ is sufficient to guarantee the existence of $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expanders. A natural question is whether such a large value of c is really necessary. As shown in Proposition 1 below, the answer is positive, at least if the expansion property applies to reasonably large subsets of U. We first prove a simple lemma, essentially a rephrasing of the definition of generalized expander, also needed for Lemma 2.

Henceforth, when referring to a $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expander, we shall assume the quantity $n/\lambda cg$ is an integer. This condition can always be met by increasing g slightly, without altering the results in any essential way.

LEMMA 1. *Let $G = (U, V, E)$ be a $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expander. If $S \subseteq U$ and $F \subseteq E$ defines a c-neighborhood $\Gamma_F(S)$ of size smaller than n/g, then $|S| \leq |\Gamma_F(S)|/\lambda c$.*

Proof. Suppose, by way of contradiction, that $|S| > |\Gamma_F(S)|/\lambda c$, that is, $|\Gamma_F(S)| < \lambda c|S|$. By Definition 1, it follows that $|S| > n/\lambda cg$. Let S' be a subset of S of size $n/\lambda cg$ and let $\Gamma_{F'}(S')$ be the c-neighborhood of S' contained in $\Gamma_F(S)$. Again by Definition 1 we have that $|\Gamma_F(S)| \geq |\Gamma_{F'}(S')| \geq \lambda cn/\lambda cg = n/g$, which is a contradiction. □

PROPOSITION 1. *If $G = (U, V, E)$ is a $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expander, with $|U| = m$, $|V| = n$, λ a fixed constant such that $0 < \lambda < 1$, and $n/cg \geq 4$, then $c = \Omega(\log(\lambda m/n)/\log g)$.*

Proof. Consider some $W \subseteq V$. We say that W *confines* $u \in U$ if $\Gamma(u) \subseteq W$. A simple combinatorial argument given in both [MV84] and [UW87] shows that the subsets of V of size w confine an average of $m\binom{n-2c+1}{w-2c+1}/\binom{n}{w} = m[w]_{2c-1}/[n]_{2c-1}$ variables apiece. (Here $[a]_b$ denotes $a(a-1)(a-2)\cdots(a-b+1)$.) In particular some set $W \subseteq V$ of size w will confine a set S with at least $\lceil m[w]_{2c-1}/[n]_{2c-1}\rceil$ variables. Choose $w = \lceil n/g - 1\rceil < n/g$ and let W and S be chosen as above. Then we have $\Gamma(S) \subseteq W$, and (by Lemma 1) $|S| \leq n/\lambda cg$; it follows that $\lceil m[w]_{2c-1}/[n]_{2c-1}\rceil \leq n/\lambda cg$. The latter inequality implies that $[n]_{2c-1}/[w]_{2c-1} > \lambda m/n$. Considering that $[n]_{2c-1} < n^{2c-1}$, and that $[w]_{2c-1} \geq (w/2)^{2c-1}$ (from $n/cg \geq 4$), we have $(2n/w)^{2c-1} > \lambda m/n$. From the definition of w, with some manipulations, we obtain $c = \Omega(\log(\lambda m/n)/\log g)$. □

One important facet of Theorem 1 lies in the introduction of the parameter g, which controls a trade-off between the number of copies and the largest set of variables for which expansion can be guaranteed. In fact the size of the smallest set of modules containing a given number of variables depends strongly on the value of c: the smaller this value the smaller the set of modules. Roughly speaking, increasing g reduces the number of copies to be processed during the simulation of a PRAM step, but creates higher congestion at memory modules, thereby making the job of accessing those copies more difficult.

Consider now a memory organization represented by a bipartite graph $G = (U, V, E)$ and let $S \subseteq U$ be a set of PRAM variables to be accessed. The copies of the variables in S will be distributed among the BDN nodes with node $v \in V$ containing $s_v \overset{\Delta}{=} |\{(u, v) : u \in S, (u, v) \in E\}|$ copies. We are interested in analyzing the effect of accessing $\min(s_v, q)$ copies from each node $v \in V$, as a function of the integer q. This operation, which we denote by Decimate(S, q), will form the basis of the memory access protocol to be described in the next section.

At any given instant, the variables of S can be partitioned into two sets: those for which at least c copies have been accessed, which we will call the *dead* variables, and the remaining ones, which we call the *live* variables. Dead variables play no further role in the simulation since enough copies have been accessed to effect either a read or write. For the live variables, more copies need to be accessed. Therefore, we are interested in bounding the size of the set of variables left alive after Decimate(S, q).

Formally, the set of copies accessed by Decimate(S, q) is represented by a set of edges $E' \subseteq E$ such that if $(u, v) \in E'$ then $u \in S$, and for each $v \in V$, there are exactly $\min(s_v, q)$ edges in E' incident upon v. Then, L is the subset of the nodes of S with degree at least c in the graph $(U, V, E - E')$.

LEMMA 2. *Suppose a memory organization has the structure of $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expander $G = (U, V, E)$. Let $k = \lceil |S|/(n/\lambda cg)\rceil$, and let $S \subseteq U$ be a set of variables. Then, if $q \geq 2k/\lambda$, the set L of variables remaining alive after* Decimate(S, q) *satisfies the relation*

$$|L| \leq \left\lceil \frac{2(1-\lambda)/\lambda}{q-2k} \right\rceil |S|. \tag{3}$$

Proof. The set S can be partitioned into k subsets $S_1, S_2, \ldots, S_k$, each of size no larger than $n/\lambda cg$. For $i = 1, 2, \ldots, k$, let $F_i \subseteq E$ be the set of edges incident upon nodes of S_i. Then F_i can be partitioned into $F_{i,1}$ and $F_{i,2}$ so that $\Gamma_{i,1} \triangleq \Gamma_{F_{i,1}}(S_i)$ is a c-neighborhood of S and $\Gamma_{i,2} \triangleq \Gamma_{F_{i,2}}(S_i)$ is a $(c-1)$-neighborhood of S. From the expansion property, we have that $|\Gamma_{i,1}| \geq \lambda c|S_i|$ and $|\Gamma_{i,2}| \geq (\lambda c - 1)|S_i|$. The latter relation is obtained considering that, by adding to $F_{i,2}$ one incident edge for each node in S_i, one obtains a c-neighborhood for S_i with at most $|S_i|$ nodes more than $\Gamma_{i,2}$ and at least $\lambda c|S_i|$ nodes in all. We now form, for each $i = 1, 2, \ldots, k$ and $j = 1, 2$, the set $F'_{i,j}$ by (arbitrarily) selecting for each node of $\Gamma_{i,j}$ one edge of $F_{i,j}$ incident upon it. Let $F = \cup_{i=1}^{k} \cup_{j=1}^{2} F'_{i,j}$. By construction, no more than $2k$ edges of F are incident upon any $v \in V$. Hence, the number of copies accessed by Decimate(S, $2k$) is at least

$$\begin{aligned}
|F| &= \sum_{i=1}^{k}\sum_{j=1}^{2} |F'_{i,j}| \\
&= \sum_{i=1}^{k}\sum_{j=1}^{2} |\Gamma_{i,j}| \\
&\geq \sum_{i=1}^{k} (\lambda c|S_i| + (\lambda c - 1)|S_i|) \\
&\geq (2\lambda c - 1)|S|.
\end{aligned}$$

Now let $H = \{v : s_v > q\}$. Each node in H contains at least $q - 2k + 1$ copies of variables in S in addition to those accessed by Decimate(S, $2k$). Considering that there is a total of $(2c-1)|S|$ copies of the variables in S, we obtain

$$(q - 2k + 1)|H| + (2\lambda c - 1)|S| \leq (2c - 1)|S|,$$

or equivalently

$$|H| \leq [2(1-\lambda)c/(q - 2k + 1)]|S|. \tag{4}$$

Given that, by assumption, $q \geq 2k/\lambda$ we have that $|H| < n/g$.

Since the copies of the live variables L not accessed by Decimate(S, q) are all in H, H must contain a c-bundle D of L. Then since $|\Gamma_D(L)| \leq |H| < n/g$, we have from Lemma 1 that $|L| \leq |H|/\lambda c$. This latter relation combined with Equation (4) yields the desired bound on $|L|$. □

We shall call the quantity $|L|/|S|$ the *survival factor* of the operation Decimate(S, q), which is the building block of the memory access protocol described in the next section. In contrast, the protocol of [UW87] is built around a procedure which, although implemented differently, has the same effect as a decimation with a constant survival factor.

Intuitively, the advantage of a smaller survival factor can be understood as follows. From standard considerations on the diameter of an n-node BDN it is clear that any implementation of decimation will require $\Omega(\log n)$ time in the worst case. However, it will be shown in the next section that $O(\log n)$ time is essentially sufficient for the execution of a Decimate(S, $O(\log n)$), as long as $|S| \leq n/(2c-1)$. From Lemma 2 we can see that, for $|S| \leq n/(2c-1)$, choosing $g = \log n$ and q a suitable multiple of g yields a survival factor of $O(1/\log n)$. The smaller the survival factor, the fewer decimations required to access all the variables. By accessing more information per unit time than the algorithm of [UW87], our algorithm succeeds in accessing the referenced variables more quickly. A similar observation is also employed in the scheme of [LPP90]. In a sense, the memory organization of [UW87] was chosen to minimize the amount of information to be extracted from any individual node in the machine. However, when related to the BDN model, this is not the dominant cost in their algorithm. Our choice of memory organization balances the work of extracting information from individual nodes with that of moving information around the machine.

4. The memory access algorithm. In this section we describe a memory access algorithm for the simulation of one step of an (n, m)-PRAM. Let λ be a constant satisfying $0 < \lambda < 1$. We assume that the memory organization is a $(\lambda, 2c-1, c, 1/\lambda cg)$-generalized expander with $c = \Theta(\log(m/n)/\log g)$. The parameter g will be specified later. It is further assumed that each node of the BDN stores in its local memory a table giving for each PRAM variable u the nodes $\gamma_1(u), \gamma_2(u), \ldots, \gamma_{2c-1}(u)$ containing copies of u. The BDN has the structure $(V, E_{\text{exp}} \cup E_{\text{AKSL}} \cup E_{\text{tree}})$, where (V, E_{exp}) is a certain expander graph to be specified later, (V, E_{AKSL}) is the graph underlying Leighton's modification of the AKS sorting network [AKS83] [Lei85], and (V, E_{tree}) is a complete binary tree with n nodes.

Let X be the set of PRAM variables to be accessed in a given PRAM step. The goal of the simulation is to access a majority of the copies of each variable in X. We shall describe the algorithm for simulating a read step, the algorithm for a write step being very similar. It is assumed that the read step is of the exclusive read variety, i.e., no two processors attempt to read the same variable. This is not an essential restriction. As already noted in several earlier studies [UW87], [AHMP87], [KU88], any concurrent read step may be reduced to an exclusive read step by means of established techniques as sketched in the Appendix.

A high level description of the memory access algorithm follows.

I. For each processor P_i generate u_i, the name of the variable it wishes to read. Let $X = \{u_1, \ldots, u_n\}$ denote the set of referenced variables.
II. For each P_i access at least c copies of u_i.
III. At P_i, select the value from the returned packet(s) with the most recent timestamp as the result of read(u_i).

Steps I and III are straightforward and are accomplished in $O(1)$ and $O(c)$ time, respectively. We focus our attention on Step II. It will be implemented as a sequence of decimations, the first one applied to the set X, and the remaining decimations each applied to the set of variables left alive by the previous decimation.

Let S be a subset of U of size at most n. The procedure Decimate(S, q) may be specified as follows: (i) S is a variable parameter denoting the set of live variables; (ii) the second argument q is a value parameter of type integer; (iii) the execution of Decimate(S, q) will cause $\min(s_v, q)$ packets destined for node v to reach that node and to be returned to their respective origins, with the appropriate values and timestamps (s_v denotes the number of copies of variables in S contained in node v); (iv) the value of S is modified by the execution of Decimate(S, q) so that, upon termination, S represents the set of live variables.

In terms of procedure Decimate, Step II of the memory access algorithm can be written as follows. Let S initially represent the set of referenced variables X.

```
for $h$ := 1 to $H$ do
        Decimate$(S, q_h)$
```

The values of $q_1, q_2, \ldots, q_H$ and H will be specified later so that after the Hth iteration S is empty, i.e., all the variables in X have been accessed.

The implementation of Decimate involves manipulating and routing sets of request packets. In the next subsection we review two routing results. The first is a *balancing* algorithm that takes a set of packets distributed among the nodes of a bounded degree network and redistributes them so that each node holds the same number. The second algorithm is a generalized routing algorithm. The implementation of Decimate in terms of these primitives is presented in the subsequent subsection. The final subsection provides an analysis of the running time of the entire memory access algorithm.

4.1. Generalized routing and balancing. The *(N, K_1, K_2)-routing problem* involves routing N packets in a network of n processors subject to the constraint that no node is the source of more than K_1 packets or the destination of more than K_2 packets. Peleg and Upfal [PU87], [PU89] formulated this problem and developed a network and a routing algorithm with an optimal number of communication steps. In the BDN model, the algorithm of [PU89] solves any instance of the (n, K_1, K_2)-routing problem in $O(\log n + K_1 + K_2 + \log(k_1 + K_2)\log n)$ time on a network with a particular expander-based structure. These results are generalized in [PU87] to handle sets of size N larger than n. In [Her91] the following result is established, which attains a lower bound of [PU87].

LEMMA 3. *There is an n-node* BDN $(V, E_{\text{exp}} \cup E_{\text{AKSL}})$ *and a deterministic algorithm that routes any instance of the (N, K_1, K_2)-routing problem in $O(\lceil N/n \rceil \log n + K_1 + K_2)$ time.*

From Lemma 3 one obtains a simple PRAM simulation scheme by distributing the PRAM variables evenly among the nodes of the BDN, with at most $\lceil m/n \rceil$ per node. Then, a write step is essentially an instance of $(n, 1, \lceil m/n \rceil)$-routing, and a read step can be implemented by means of a $(n, 1, \lceil m/n \rceil)$-routing step followed by an $(n, \lceil m/n \rceil, 1)$-routing step. These observations lead to the following result.

PROPOSITION 2. *An arbitrary step of a* PRAM *with n processors and $m \geq n$ memory cells can be simulated by the* BDN $(V, E_{\text{exp}} \cup E_{\text{AKSL}})$ *in $O(\log n + \min(n, m/n))$ time.*

In essence, the algorithm of [PU89] (as well as its refinement in [Her89]) reduces an instance of the generalized routing problem to a partial permutation routing problem by means of *balancing* (referred to as *token distribution* in [PU89]). Balancing involves redistributing a set of packets initially scattered unevenly among the nodes of a BDN so that each node has the same number. Formally, a *distributed set* $\hat{Z} \overset{\Delta}{=} (Z, f)$ is a set Z of packets distributed among the nodes of the BDN with packet $z \in Z$ at node $f(z)$. The *degree* $\Delta(\hat{Z})$ of $\hat{Z}$ is the maximum number of packets assigned by f to any one node. The set $\hat{Z}$ is *balanced* if it has degree $\lceil |Z|/n \rceil$. A distributed set $\hat{Z} = (Z, f)$ of packets, labeled with integer values, is said to be *balancesorted* if it is balanced and the packets with the $\lceil |Z|/n \rceil$ smallest keys lie in node 1, the next $\lceil |Z|/n \rceil$ next smallest lie in node 2, and so on. The following result is established in [Her91].

LEMMA 4. *There exists an n-node* BDN $(V, E_{\text{exp}} \cup E_{\text{AKSL}})$ *and a deterministic algorithm that balancesorts any distributed set $\hat{Z} = (Z, f)$ in $O(\lceil |Z|/n \rceil \log n + \Delta(\hat{Z}))$ time.*

4.2. Implementation of Decimate. The algorithm outlined in this subsection is based on the following primitives: sorting, parallel prefix, balancing, and the generalized routing mentioned in the previous subsection. The underlying network has the structure $(V, E_{\text{exp}} \cup E_{\text{AKSL}} \cup E_{\text{tree}})$ and was chosen to support these primitives as efficiently as possible. The algorithm can be adapted to any other architecture that supports these primitives.

Recall that at any given point in the computation S represents the set of live variables. The overall idea is as follows. For each $v \in V$, we *select* exactly $d_v \overset{\Delta}{=} \min(s_v, q)$ of the s_v requests directed at module v. The selected packets are routed to their respective destinations, where they are loaded with the appropriate values and then routed back to their origins. The details follow.

```
Decimate($S, q$)
1. For each $u_i \in S$ and each $j \in \{1, \dots, 2c - 1\}$ generate a request packet containing
   an origin $P_i$, a destination $\gamma_j(u_i)$, the label $u_i$ itself, and fields to store a value and a
   timestamp of the copy of $u_i$ contained in node $\gamma_j(u_i)$.
2. Sort the request packets by destination.
3. Select $d_v$ request packets from those with destination $v$; discard the others.
4. Route the selected request packets to the corresponding nodes.
5. Load values and timestamps into request packets.
6. Route the selected requests (now satisfied) back to their origins.
7. Let each origin $P_i$ count the number of packets received during Step 6 and, if they are
   at least $c$ in number, mark $u_i$ dead and remove the corresponding packets from the set
   of request packets.
```

LEMMA 5. *Decimate*(S, q) *can be implemented on the* BDN $(V, E_{\text{exp}} \cup E_{\text{AKSL}} \cup E_{\text{tree}})$ *with running time* $O(\lceil c|S|/n \rceil \log n + c + q)$.

Proof. First, Steps 1, 5, and 7 are sequential computations and can be performed locally at each node in $O(c)$, $O(q)$, and $O(c)$ time, respectively.

Step 2 consists of balancing and sorting a distributed set of size no larger than $(2c - 1)|S|$ and with at most $2c - 1$ elements per node. By Lemma 4, this operation takes $O(\lceil c|S|/n \rceil \log n + c)$ time.

Step 3 can be reduced to a parallel prefix computation [KRS85] on a sequence of length $(2c - 1)|S|$ as follows. Following Step 2, the packets are arranged in increasing order of destination, and so those with a common destination v form a contiguous subsequence σ_v. Using parallel prefix it is possible to number the packets in sequence σ_v with the numbers $1, \dots, s_v$. Those with numbers at most q are selected; the other are discarded. All of this can be executed on (V, E_{tree}) in $O(\lceil c|S|/n \rceil + \log n)$ time.

Step 4 involves routing $(2c - 1)|S|$ packets with at most $2c - 1$ per source and at most q per destination. This is an instance of the $((2c - 1)|S|, 2c - 1, q)$-routing problem, which by Lemma 3 takes $O(\lceil c|S|/n \rceil \log n + c + q)$ time. Similarly, Step 6 is an instance of the $((2c - 1)|S|, q, c)$-routing problem and has the same running time.

The contributions of the various steps add up to the stated running time. □

4.3. Simulation time. In this subsection we establish the following result.

THEOREM 2. *An arbitrary step of an* (n, m)-PRAM *with* $m \geq n$ *can be simulated by the* BDN $(V, E_{\text{exp}} \cup E_{\text{AKSL}} \cup E_{\text{tree}})$ *in* $O((\log n \log m)/ \log \log n)$ *time.*

Proof. From Proposition 2, if $m \leq (n \log n)/(3\lambda g_0(\lambda))$, then the simulation can be done in $O(\log n)$ time, and Theorem 2 is established. Therefore, we can assume that $m > (n \log n)/(3\lambda g_0(\lambda))$, and thus we can choose $g = \log n$ without violating the constraint of Theorem 1 that $g \leq 3\lambda g_0(\lambda) m/n$.

Also, if $m \geq (n/3\lambda)2^{n(1-\lambda)/4}$, then $\log m = \Omega(n)$ and Theorem 2 again follows from Proposition 2. Therefore, we can assume that $m < (n/3\lambda)2^{n(1-\lambda)/4}$. For $g = \log n$ and n large enough (such that $\log n \geq 2g_0(\lambda)$), the upper bound on m guarantees that the condition $2c - 1 \leq n$ of Theorem 1 is satisfied, so that we are entitled to assume the existence of the generalized expander underlying our simulation.

We have already seen that Steps I and III of the memory access algorithm can be completed in $O(c)$ time, so we now turn our attention to the running time of Step II, which consists of a sequence of H decimations with parameters $q_1, q_2, \dots, q_H$. Let L_h be the set of live variables after the hth decimation.

For the first decimation, applied to the set S of PRAM variables ($|S| \leq n$), we choose $q_1 = \lceil (4/\lambda) \lceil \lambda c g \rceil \rceil + 1$. Then, $q_1 \geq 2k/\lambda$ with $k = \lceil (|S|/n) \lambda c g \rceil \leq \lceil \lambda c g \rceil$, and by Lemma 2, $|L_1| \leq n/\lambda c g$. By Lemma 5, this decimation runs in $O(cg + c \log n)$ time.

For the remaining decimations, we choose $q_2 = q_3 = \cdots = q_H = \lceil g2(1-\lambda)/\lambda + 2\rceil$. By Lemma 2, considering that $k = \lceil(|L_h|/n)\lambda cg\rceil \leq \lceil(|L_1|/n)\lambda cg\rceil \leq 1$, for $h \geq 2$ we have $|L_h| \leq |L_{h-1}|/g$. Thus, for $h \geq 2$, $|L_h| \leq |L_1|/g^{h-1} \leq n/\lambda cg^h$. Choosing $H = \lceil \log n/\log g\rceil + 1$ ensures that $L_H = \emptyset$.

By Lemma 5, the hth decimation applied to a set of at most $|L_{h-1}|$ variables runs in time $O(\log n + c + g)$ for each $h \geq 2$.

The total time for Step II, and indeed for the entire simulation, is $O(cg + c\log n + (\log n + c + g)(\log n/\log g))$. Using the choice $g = \log n$ and recalling that $c = \Theta(\log(m/n)/\log g)$, we arrive at the result claimed above. □

5. Lower bound. We begin by stating the assumptions that underlie the known lower bounds as well as the refinement presented here. For a more extended discussion of some points the reader is referred to [AHMP87].

The (n, m)-PRAM to be simulated is of the exclusive-read exclusive-write (EREW) variety, that is, concurrent access to the same memory cell by two different processors is disallowed. Lower bounds for the EREW PRAM clearly apply if concurrent access is allowed.

The PRAM computation to be simulated is an arbitrary sequence of PRAM instructions. Each PRAM instruction consists of either an arbitrary n-tuple of reads or an arbitrary n-tuple of writes. In the lower bound arguments, it is essential that a sequence of $\Omega(m/n)$ instructions be considered and that the addresses to be accessed at each instruction be chosen independently of those in other instructions. In this sense, the sequence is straight-line. The lower bounds do not necessarily apply to sequences generated by running a fixed-sized program on a large enough input. Throughout this section the term *instruction* will be used to refer to the individual steps of a PRAM computation. The term BDN step will refer to a single machine cycle of the underlying n-node BDN.

The simulation is *on line*, in the sense that each PRAM instruction is made known to the BDN only after the simulation of the previous read instructions has been completed.

As for the memory organization, it is assumed that, at any given time, for each PRAM variable u there is a set $C(u)$ of BDN memory cells where the current (last written) value of u is available. The set $C(u)$ is allowed to change with time, even during the simulation of instructions that do not write u. If $u \neq u'$, it is assumed that $C(u)$ and $C(u')$ do not overlap. Thus, at a given time a memory cell can hold the value of at most one variable. The cells in $C(u)$ are sometimes referred to as the valid copies of u.

The memory access protocol is *point to point* in the following sense. If a PRAM processor writes a variable u, then the corresponding BDN node must send a distinct message (with the value to be assigned to u) to each of the BDN nodes containing a cell in $C(u)$. The crucial restriction is that messages directed to different nodes can not be combined, even if they share a portion of their paths. No such assumption is made for read instructions (the lower bound is based only on the fact that at least *one* valid copy must be accessed for each variable read).

The following result was established independently in [AHMP87] and [KU88]:

THEOREM 3. *For all real constants $\epsilon > 0$, and $\epsilon' > 0$, if $m = \Omega(n^{2+\epsilon})$ and $T \geq (1+\epsilon')\lceil m/n\rceil$, then the worst-case simulation time for a straight-line program running for T steps on an* EREW (n, m)-PRAM *is*

$$\Omega\left(T \min\left[\sqrt{n\log n}, \frac{\log m \log n}{\log\log m}\right]\right),$$

for any on-line point-to-point simulation of the PRAM *by an n-processor bounded degree network.*

Under the same assumption, by refining the proof technique of [AHMP87], [KU88], we can extend the lower bound to the range $n \leq m \leq n^{2+\epsilon}$.

THEOREM 4. *For all real constants $\epsilon > 0$, and $\epsilon' > 0$, if $2n < m \leq n^{2+\epsilon}$ and $T \geq (1+\epsilon')\lceil m/n \rceil$, then the worst-case simulation time for a straight-line program running for T steps on an* EREW (n, m)-PRAM *is*

$$\Omega\left(T\frac{\log^2(m/n)}{\log\log(m/n)}\right),$$

for any on-line point-to-point simulation of the PRAM *by an n-processor bounded degree network.*

We first review the ideas behind the proof of Theorem 3 and then indicate the modifications necessary to obtain Theorem 4. For ease of reference, we will use the same notation as in [AHMP87].

The key idea of the approach is the notion of redundancy defined below. Let $G = (V, E)$ be the graph representing the structure of the network. Let d denote the maximum degree of the nodes in the underlying BDN, and let ρ denote the quantity $\lfloor \log(m/2n)/(2\log d) \rfloor - 1$. For each node $v \in V$, let $B(v)$ represent the set $\{x : \delta(v,x) \leq \rho\}$ of nodes where δ is the usual graph-theoretic distance.

Let Γ_u^t denote the set of nodes which contain a valid copy of the variable u at the start of the simulation of the tth instruction of the PRAM program. The *redundancy* of a variable u is defined as follows:

$$r_u^t \triangleq \min_{v \in V} |\Gamma_u^t / B(v)|.$$

The average redundancy over the entire set of variables is $r^t \triangleq \sum_{u \in U} r_u^t / m$.

Intuitively, the average redundancy measures the "accessibility" of the variables: the lower the average redundancy, the harder it will be to simulate read instructions. The following lemma gives a quantitative formulation of this phenomenon.

LEMMA 6. *If $n \leq m \leq n^2$, and the average redundancy before the simulation of a read instruction is r, then there exists a set of n variables that will require $g(r) = \Omega((m/2n)^{1/(8r+2)})$* BDN *steps to read.*

Proof. First observe that if $m < 4d^4 n$, then we are claiming that $g(r) = \Omega(1)$, which is clearly true. Thus we may assume that $m \geq 4d^4 n$, which implies $\rho = \lfloor \log(m/2n)/(2\log d) \rfloor - 1 \geq 1$. Also, $|B(v)| \leq d^{\rho+1} \leq (m/2n)^{1/2}$, for all $v \in V$. Let Γ_x be the set of BDN nodes that contain a valid copy of variable $x \in U$ at the beginning of the instruction being considered, and let $r_x = \min_{v \in V} |\Gamma_x / B(v)|$ be the redundancy of x. Let $U' = \{x \in U : r_x \leq 2r\}$. Clearly $|U'| \geq m/2$. Partition U' into disjoint sets X_v, $v \in V$, so that $X_v \subseteq \{x \in U' : r_x = |\Gamma_x / B(v)|\}$. (Note that some of the X_v may be empty.) We will deal separately with the case $r < \frac{1}{2}$ and the case $r \geq \frac{1}{2}$.

If $r < \frac{1}{2}$, then all the elements of U' have redundancy 0, since each individual variable has redundancy strictly less than one and individual variables can only have integral redundancies. Consider a ball $B(v)$ such that $|X_v|$ is as large as possible, which implies that $|X_v| \geq \lceil m/2n \rceil$. Reading the $\lceil m/2n \rceil$ variables in X_v will clearly require at least

$$\frac{|X_v|}{|B(v)|} \geq \frac{\lceil m/2n \rceil}{(m/2n)^{1/2}} = \Omega((m/2n)^{1/2})$$

steps.

Now suppose that $r \geq \frac{1}{2}$. Without loss of generality we assume that $r \leq (\log(m/2n) - 2)/8$, since otherwise the bound claimed for $g(r)$ is $g(r) = \Omega(1)$ which is obvious. Let $L \subseteq V$ be the set of indices of the $\lceil n/(m/2n)^{1/2+1/8r} \rceil$ largest X_v, and let $X_L = \cup_{v \in L} X_v$. Clearly

$|X_L| \geq (m/2n)|L| \geq n(m/2n)^{1/2-1/8r}$. We also let $B_L = \cup_{v \in L} B(v)$. By the definition of L above, $|L|$ can be expressed in the form $|L| = n/(m/2n)^{1/2+1/8r} + \alpha$, for some real α satisfying $0 \leq \alpha < 1$. Thus

$$|B_L| \leq |L|(m/2n)^{1/2} = \frac{n}{(m/2n)^{1/8r}} + \alpha(m/2n)^{1/2} < \frac{2n}{(m/2n)^{1/8r}},$$

where we have made use of the relations $\alpha < 1, r \geq \frac{1}{2}$, and $m \leq n^2$.

Now for any $W \subseteq V$, let $A_W = \{x \in X_L \ : \ \Gamma_x \subseteq B_L \cup W\}$. Recall that each variable in U' has redundancy at most $2r$ and hence at most $\lfloor 2r \rfloor$. A simple combinatorial argument given in [AHMP87] establishes the following: for any k with $\lfloor 2r \rfloor \leq k \leq n$, there is a subset W of V with $|W| = k$ and

$$|A_W| \geq |X_L| \binom{n - \lfloor 2r \rfloor}{k - \lfloor 2r \rfloor} \Big/ \binom{n}{k} \geq |X_L| \left(\frac{k - \lfloor 2r \rfloor + 1}{n} \right)^{\lfloor 2r \rfloor}.$$

We now choose

$$k = \left\lceil \lfloor 2r \rfloor - 1 + n(m/2n)^{-(1/2-1/8r)(1/\lfloor 2r \rfloor)} \right\rceil.$$

Straightforward manipulations making use of the assumptions $m \leq n^2$ and $r \leq (\log(m/2n) - 2)/8$ show that $k \leq n$. Use of the chosen value of k in the lower bound for $|A_W|$ yields $|A_W| \geq n$. Then we can choose the set of variables to be read in the current instruction as any n-element subset of A_W. Since all of the valid copies of these variables are contained in $|B_L \cup W| \leq |B_L| + |W|$ nodes, it follows that the simulation of that read instruction (in the case where $r \geq \frac{1}{2}$) will require at least

$$\frac{n}{|B_L| + k} = \Omega\left(\min\left(\frac{n}{2r}, \left(\frac{m}{2n}\right)^{(\frac{1}{2} - \frac{1}{8r})\frac{1}{\lfloor 2r \rfloor}}, \quad \left(\frac{m}{2n}\right)^{1/8r} \right) \right) = \Omega\left(\left(\frac{m}{2n}\right)^{1/8r} \right)$$

time. Combining the results for the two subcases $r < \frac{1}{2}$ and $r \geq \frac{1}{2}$, we obtain the stated result. □

A similar lemma, with a different form of g, was obtained in [AHMP87], [KU88] for $m = \Omega(n^{2+\epsilon})$. The two key refinements of the proof technique that enable its application to the range of m in Lemma 6 are the following: (i) the value of ρ is chosen as a function of both m and n, instead of just n, and (ii) in constructing the set of variables that are hard to read, the union of several balls is considered, rather than just one ball.

The next lemma relates the average redundancy to the time spent to simulate the write instructions.

LEMMA 7. *Consider an interval during which an n-node* BDN *simulates a sequence of* (n, m)-PRAM *instructions such that each of the m variables is written at least once. The average redundancy r at the end of the interval and the length t of the interval satisfy the relation*

$$t \geq \rho r m/n. \tag{5}$$

Proof. As a simple consequence of its definition, the redundancy r_u of variable u at the end of the interval is at most the number of copies that are at a distance greater than ρ from the last processor that has written u. Due to the point-to-point assumption, at least ρr_u processor steps can be charged to variable u. Then, (5) follows from the observations that $rm = \sum_{u \in U} r_u$, and that at most tn processor steps are executed in the given interval. □

Proof of Theorem 4. We assume throughout the proof that $2n \leq m \leq n^2$, so that we can invoke Lemma 6. The result for the range $n^2 \leq m \leq n^{2+\epsilon}$, with ϵ a fixed positive constant, trivially follows from the case $m = n^2$.

We define an *adversary program* consisting of $\lceil m/n \rceil + \tau$ PRAM instructions. The first $\lceil m/n \rceil$ are write instructions that initialize all the PRAM variables. The remaining are read instructions chosen according to Lemma 6 so that each require at least $g(r)$ BDN steps to simulate.

Let t_i be the time when the BDN begins to simulate the ith read instruction, assuming that the simulation of the adversary program starts at time 1 and ends at time T_{sim}. Let r^i be the average redundancy at time t_i. Finally, let

$$\begin{aligned} r^* &= \left(\frac{\log(m/2n)}{2\log\log(m/2n) - \log\log\log(m/2n)} - 2 \right) /8 \\ &= \Omega(\log(m/2n)/\log\log(m/2n)), \end{aligned}$$

and observe that

$$g(r^*) = \Omega\left((m/2n)^{1/(8r^*+2)}\right) = \Omega\left(\log^2(m/2n)/\log\log(m/2n)\right).$$

We consider two complementary cases:

1. For some $i \in \{1, 2, \ldots, \tau\}$, $r^i \geq r^*$. Then, from Lemma 7,

$$T_{\text{sim}} \geq t_i \geq \rho r^* m/n = \Omega((m/n)\log^2(m/2n)/\log\log(m/2n)). \tag{6}$$

2. For all $i \in \{1, 2, \ldots, \tau\}$, $r^i < r^*$. Then, from Lemma 6, each read instruction requires at least $g(r^*)$ steps to simulate; hence,

$$T_{\text{sim}} = \Omega(\tau \log^2(m/2n)/\log\log(m/2n)). \tag{7}$$

If $\tau = \lceil \epsilon' m/n \rceil$ with $0 < \epsilon' < 1$, then Theorem 4 follows from the combination of (6) and (7).

The extension to any $\epsilon' > 1$ is easily obtained by applying the above considerations to the concatenation of $\lfloor (1+\epsilon')/2 \rfloor$ blocks, each with the structure of the $2\lceil m/n \rceil$ instruction adversary program defined above. □

A comparison of the lower bound of Theorem 4 with the upper bound provided by Theorem 2 shows that the simulations of §4 are optimal whenever $\Omega(n^{1+\epsilon}) \leq m \leq O(2^{(\log n)^{\alpha}})$ for some positive constants ϵ and α, as claimed in the introduction.

It should be noted that, for $m < n2^{\sqrt{O(\log n \log\log n)}}$, the lower bound of Theorem 4 is weaker than the straightforward $\Omega(T \log n)$ lower bound based on diameter considerations.

6. Conclusions. In this paper optimal point-to-point simulations have been given for a wide range of the memory size m. It is natural to try to close the gap between upper and lower bounds for the remaining values of m. Other important problems related to deterministic PRAM simulations on BDNs are the explicit construction of generalized expanders for the memory organization, the characterization of the complexity of non–point-to-point memory access protocols, and the development of simulations on BDNs with a simpler structure than those proposed in this paper, possibly avoiding the AKS network.

7. Appendix. The techniques of §4 applied to the problem of simulating a step of an exclusive read exclusive write PRAM in which the referenced variables are assumed to be distinct. In this appendix we extend these techniques to handle the concurrent access PRAMs.

In a concurrent read concurrent write PRAM a number of processors may read, or attempt to write to, the same variable in a single PRAM step. For the purposes of this appendix we will assume that in the event of concurrent writes the lowest numbered processor succeeds. The techniques outlined below may be modified to handle other variants of concurrent write PRAMs without too much difficulty. Techniques similar to those outlined here have been employed (implicitly or explicitly) in most of the earlier studies cited in the Introduction (see also [Vis83]). (The terminology of leaders and followers used below is borrowed from [LPP90], which contains a detailed description of the reduction of concurrent access steps to exclusive access steps for the mesh of trees architecture.)

We focus initially on the problem of concurrent read PRAMs and discuss concurrent write PRAMs at the end of the section. Let u_i denote the PRAM variable referenced by the ith PRAM processor P_i^{PRAM}. The variables $u_1, \ldots, u_n$ need not be distinct. To complete this PRAM step a copy of the value of variable u_i must be routed to the ith BDN processor P_i which simulates P_i^{PRAM}. This may be accomplished as follows.

1. Let P_i generate a token labeled with u_i and P_i.

2. For each referenced variable choose one token from among those that refer to it as a *leader*, all other tokens referring to that variable are *followers*.

3. Load each leader token with the value of the corresponding variable.

4. Distribute the value of each leader to its followers and return all tokens to the processors that generated them.

Step 1 requires $O(1)$ time to complete.

Step 2 may be implemented as follows.

(i) Sort the tokens by variable name. (Tokens referring to the same variable are ordered by processor number.)

(ii) For each processor P_i compare the token it holds with that held by P_{i-1}. If the two tokens refer to the same variable let P_i mark its token as a follower; otherwise mark it as a leader. (The token held by P_1 should be marked as a leader.)

Once the tokens have been sorted by variable name, all those referring to the same variable are grouped together in consecutively numbered processors. The token in the lowest numbered processor in each group is chosen as the leader and the others are marked as its followers. The sorting step involves sorting n packets and can be completed in $O(\log n)$ time. The comparison of adjacent tokens will involve a permutation routing step which can also be completed in $O(\log n)$ time.

For each referenced variable there is exactly one leader token. Thus Step 3 is essentially an exclusive read step and so can be completed in $O(\log n \log m / \log\log n)$ time by the results of §4.

Note that following the completion of Step 3 the tokens occupy the same processors they occupied before that step. At this point each leader token is labeled with the value of the appropriate variable which must now be distributed to the followers of that token. Each group of tokens (a leader and its followers) corresponding to a particular variable occupy a contiguous group of processors $P_a, \ldots, P_b$ where P_a holds the leader. Assign a value v_i to each processor as follows. If the token at that processor is a leader then v_i is the value loaded into that token during Step 3; otherwise the v_i is taken to the zero. For each group of processors $P_a, \ldots, P_b$ compute the quantities $w_i = \sum_{k=a}^{i} v_k$ and deposit each w_i in processor P_i. This can be accomplished using a prefix computation as shown in [Sch80] (p. 491) (where the problem is known as the summing-by-groups problem). This step also requires $O(\log n)$ time. It can easily be seen that each w_i within a group is equal to v_a the value of the leader of that group. The various tokens can be routed back to the processors that originated them in $O(\log n)$ time using a permutation routing step.

Summing the contribution of the various steps, it can be seen that any concurrent read PRAM step can be completed in $O(\log n \log m / \log\log n)$ time.

To simulate a concurrent write step execute Steps 1 to 3 of the above with the following modifications. First, each token is marked with the value to be written to the variable in question. Step 3 is an exclusive write step. By the reasoning given above it can be seen that all three steps can be concluded in $O(\log n \log m / \log\log n)$ time.

Acknowledgments. We are greatly indebted to the two referees for their patience and thoroughness in reading an earlier error-filled version of this manuscript. Their numerous valuable suggestions have greatly enhanced the readability of this paper. We also would like to thank Paul Bay for his comments on an earlier version of the manuscript.

REFERENCES

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata, *Deterministic simulation of idealized parallel computers on more realistic ones*, SIAM J. Comput., 16 (1987), pp. 808–835.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi, *An $O(n \log n)$ sorting network*, in Proc. of the 15th Annual Symposium on Theory of Computing, Boston, MA, 1983, pp. 1–9.

[BH85] A. Borodin and J. E. Hopcroft, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 33 (1985), pp. 130–145.

[FW78] S. Fortune and J. Wyllie, *Parallelism in random access machines*, in Proc. of the 10th Annual Symposium on Theory of Computing, San Diego, CA, 1978, pp. 114–118.

[Gif79] D. K. Gifford, *Weighted voting for replicated data*, in Proc. of the 7th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, 1979, pp. 150–159.

[Gol78] L. Goldschlager, *A unified approach to models of synchronous parallel machines*, in Proc. of the 10th Annual Symposium on Theory of Computing, San Diego, CA 1978, pp. 89–94. Also appears in J. Assoc. Comput. Mach. 29 (1982), pp. 1073–1086.

[Her89] K. T. Herley, *Efficient simulations of small shared memories on bounded degree networks*, in Proc. of the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 390–395.

[Her90] ———, *Space-efficient representations of shared data for parallel computers*, in Proc. of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 407–416.

[Her91] ———, *A note on the token distribution problem*, Inform. Process. Lett. 38 (1991), pp. 329–334.

[HMP89] T. Hagerup, K. Mehlorn, and B. Parker, Unpublished notes communicated by K. Mehlhorn.

[HP89] S. W. Hornick and F. P. Preparata, *Deterministic PRAM simulation with constant redundancy*, in Proc. of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM 1989, pp. 103–109.

[KR90] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, J. Van Leeuwen ed., North-Holland, Amsterdam, 1990.

[KRS85] C. P. Kruskal, L. Rudolph, and M. Snir, *The power of parallel prefix*, IEEE Trans. Comput. c-34 (1985), pp. 965–968.

[KU88] A. R. Karlin and E. Upfal, *Parallel hashing: an efficient implementation of shared memory*, J. Assoc. Comput. Mach., 35 (1988), pp. 876–892.

[Kuc82] L. Kucera, *Parallel computation and conflicts in memory access*, Inform. Process. Lett. 14 (1982), pp. 93–96.

[Lei84] F. T. Leighton, *New lower bound techniques for VLSI*, Math. Systems Theory, 17 (1984), pp. 47–70.

[Lei85] ———, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput., c-34 (1985), pp. 344–354.

[LMR88] F. T. Leighton, B. Maggs, and S. Rao, *Universal packet routing algorithms*, in Proc. of the 29th Annual Symposium on Foundations of Computer Science, White Plains, NY 1988, pp. 256–269.

[LPP88] F. Luccio, A. Pietracaprina, and G. Pucci, *A probabilistic simulation of PRAMs on a bounded degree network*, Inform. Process. Lett., 28 (1988), pp. 141–147.

[LPP90] ———, *A new scheme for the deterministic simulation of PRAMs in VLSI*, Algorithmica, 5 (1990), pp. 529–544.

[LPS86] A. LUBOTZKY, R. PHILLIPS, AND P. SARNAK, *Explicit expanders and the Ramanujan conjectures*, in Proc. of the 18th Annual Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 240–246.

[LPS88] A. LUBOTZKY, R. PHILLIPS, AND P. SARNAK, *Explicit expanders and the Ramanujan conjectures*, Combinatorica, 8 (1988), pp. 261–277.

[LPV81] G. F. LEV, N. PIPPENGER, AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., c-30 (1981), pp. 93–100.

[MV84] K. MEHLHORN AND U. VISHKIN, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories*, Acta Inform. 21 (1984), pp. 339–374.

[NMB83] D. D. NATH, S. N. MAHESHWARI, AND P. C. P. BHATT, *Efficient VLSI networks for parallel processing based on orthogonal trees*, IEEE Trans. Comput., c-32 (1983), pp. 569–581.

[PU87] D. PELEG AND E. UPFAL, *The generalized packet routing problem*, Theoret. Comput. Sci., 53 (1987), pp. 281–293.

[PU89] ———, *The token distribution problem*, SIAM J. Comput., 18 (1989), pp. 229–243.

[Ran91] A. G. RANADE, *How to emulate shared memory*, J. Comput. System Sci., 42 (1991), pp. 307–326.

[Sch80] J. SCHWARTZ, *Ultracomputers*, ACM Trans. Program. Languages Systems, 2 (1980), pp. 484–521.

[Sni85] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.

[Sny86] L. SNYDER, *Type architectures, shared memory and the corollary of modest potential*, in Annual Review of Computer Science, Vol. 1, J. F. Traub et al., eds., 1986, pp. 289–317.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.

[Tho79] R. H. THOMAS, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Trans. Database Systems, 4 (1979), pp. 180–209.

[Upf84] E. UPFAL, *A probabilistic relation between desirable and feasible models of parallel computation*, in Proc. of the 16th Annual ACM Symposium on Theory of Computing, Washington, D.C., 1984, pp. 258–265.

[UW87] E. UPFAL AND A. WIGDERSON, *How to share memory in a distributed system*, J. Assoc. Comput. Mach., 34 (1987), pp. 116–127.

[Val88] L. G. VALIANT, *Optimally universal parallel computers*, Philos. Trans. Roy. Soc. London Ser. A, 326 (1988), pp. 373–376.

[Vis83] U. VISHKIN, *Implementation of simultaneous memory address access in models which forbid it*, J. Algorithms, 4 (1983), pp. 45–50.

[Vis84] ———, *A parallel-design distributed-implementation (PDDI) general-purpose computer*, Theor. Computer Sci., 32 (1984), pp. 157–172.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 293–312, April 1994

LOWER BOUNDS FOR RANDOMIZED k-SERVER AND MOTION-PLANNING ALGORITHMS*

HOWARD KARLOFF†, YUVAL RABANI‡, AND YIFTACH RAVID‡

Abstract. In this paper, the authors prove lower bounds on the competitive ratio of randomized algorithms for two on-line problems: the k-server problem, suggested by Manasse, McGeoch, and Sleator [*Competitive algorithms for on-line problems*, J. Algorithms, 11 (1990), pp. 208–230], and an on-line motion-planning problem due to Papadimitriou and Yannakakis [*Shortest paths without a map*, Lecture Notes in Comput. Sci. 372, Springer-Verlag, New York, 1989, pp. 610–620]. The authors prove, against an oblivious adversary,

1. an $\Omega(\log k)$ lower bound on the competitive ratio of any randomized on-line k-server algorithm in any sufficiently large metric space,
2. an $\Omega(\log \log k)$ lower bound on the competitive ratio of any randomized on-line k-server algorithm in any metric space with at least $k + 1$ points, and
3. an $\Omega(\log \log n)$ lower bound on the competitive ratio of any on-line motion-planning algorithm for a scene with n obstacles.

Previously, no superconstant lower bound on the competitive ratio of randomized on-line algorithms was known for any of these problems.

Key words. competitive analysis, k-server problem, on-line algorithm, motion planning

AMS subject classifications. 68Q25, 93C85

1. Introduction.

On-line algorithms handle sequences of events, each event being handled before future events are known. Among the on-line problems recently studied are paging (Sleator and Tarjan [ST]), on-line vertex coloring (Lovász, Saks, and Trotter [LST]), metrical task systems (Borodin, Linial, and Saks [BLS]), the k-server problem (Manasse, McGeoch, and Sleator [MMS]), layered graph traversal (Baeza-Yates, Culberson, and Rawlins [BCR] and Papadimitriou and Yannakakis [PY]), and on-line motion-planning [PY].

Sleator and Tarjan [ST] suggested comparing on-line algorithms not to each other but to an optimal off-line algorithm that knows the entire sequence in advance. This approach is called *competitive analysis*. We say a (randomized) on-line algorithm A is *c-competitive* if there is a constant a dependent on the initial configuration but independent of the event sequence so that, for all event sequences σ, the (expected) cost incurred by A on σ is at most a plus c times the optimal cost to handle σ. The infimum of all c such that A is c-competitive is A's *competitive ratio*.

In the case of a randomized algorithm it is important to define accurately the power of the adversary (see Raghavan and Snir [RS] and Ben-David et al. [BBKTW]). The *adaptive off-line* adversary may adapt the sequence of requests it produces to the random choices made to date by the on-line algorithm and then pay for the entire sequence optimally. Ben-David et al. show that randomization against this adversary does not improve on-line performance compared with deterministic algorithms [BBKTW]. The *adaptive on-line* adversary adapts the request sequence to the on-line algorithm's random choices. However, it must serve each request before the on-line algorithm serves it, and therefore the cost of this adversary might be far from optimal. Ben-David et al. show that randomization against this adversary cannot help much. If there exists a c-competitive randomized algorithm against an adaptive on-line adversary, then there exists a c^2-competitive deterministic algorithm [BBKTW]. This paper

*Received by the editors January 15, 1992; accepted for publication (in revised form) November 6, 1992.

†Department of Computer Science, University of Chicago, Chicago, Illinois 60637. This author was supported in part by National Science Foundation grant CCR 8807534.

‡Computer Science Department, School of Mathematics, Tel-Aviv University, Tel-Aviv 69978, Israel.

deals with the *oblivious* adversary, the weakest of the three and the "traditional" adversary. In contrast to adaptive adversaries, the oblivious adversary must fix the sequence of requests in advance and then pay for it optimally. Randomization can be used by the on-line algorithm to "hide" its choices from the oblivious adversary.

We study the k-server and motion-planning problems. In the *k-server problem*, k servers move among the points of a metric space $\mathcal{M}$, serving requests. A *request* is a point of $\mathcal{M}$. To *serve* the request means to move a server to the request site. The algorithm pays a price equal to the distance moved. Requests are served *on-line*. This means that each request is served before future requests are known. Manasse et al. proved that in every metric space on at least $k+1$ points there is a lower bound of k on the competitive ratio of any deterministic on-line k-server algorithm [MMS]. This lower bound is also applicable for randomization against adaptive adversaries but not for randomization against an oblivious adversary. Manasse et al. also conjectured that for every metric space and every k, there is a deterministic k-competitive on-line algorithm [MMS]. They proved their conjecture for $k = 2$ and for $k = n - 1$, where n is the cardinality of the (finite) metric space. This conjecture was proven for uniform metric spaces [ST] and for the infinite metric space that is isomorphic to the real line by Chrobak et al. [CKPV]. Chrobak and Larmore [CL] generalized the result to infinite metric spaces that are isomorphic to trees. Deterministic competitive algorithms for all metric spaces and all k were given by Fiat, Rabani, and Ravid [FRR]. Grove [Ge] proved that the so-called Harmonic randomized server algorithm, suggested by Raghavan and Snir [RS], is competitive for all k.

A searcher in the on-line motion-planning model of Papadimitriou and Yannakakis [PY] is a point particle that starts at a point s in the Euclidean plane and moves to a known target. In our version of the problem, the target is a vertical line, and the searcher need only reach a point of his own choosing on it. However, the plane is peppered with stationary open rectangular obstacles that are disjoint from each other and from the source s and target t. Each rectangle has integral side lengths. The searcher can "see" only those obstacles that are connected by an obstacle-free line segment to the searcher's position. We compare the cost incurred by the searcher to the length of a shortest obstacle-free source-target path, which is the cost incurred by an optimal algorithm that sees the entire scene. Papadimitriou and Yannakakis gave an $\Omega(\sqrt{n})$ lower bound on the ratio between the cost of a deterministic on-line algorithm and the optimal cost in a scene containing n rectangles [PY].

The use of randomization against an oblivious adversary has indeed led to superior algorithms. Fiat et al. exhibited a randomized paging algorithm with a competitive ratio bounded by $2\mathcal{H}_k$, where $\mathcal{H}_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k} \sim \ln k$ [FKLMSY] (a different algorithm was subsequently proven $\mathcal{H}_k$-competitive by McGeoch and Sleator [MS]). This is to be contrasted with the lower bound of k for a deterministic paging algorithm. Fiat et al. also proved a lower bound of $\mathcal{H}_k$ for the competitive ratio of randomized paging algorithms [FKLMSY]. They conjectured that $\mathcal{H}_k$ is an upper bound for all metric spaces and all k. This conjecture was disproved by Karlin et al. [KMMO], who exhibited a family of 3-point metric spaces and a lower bound approaching $\frac{e}{e-1} > 1.5 = \mathcal{H}_2$ for the competitive ratio of any randomized 2-server algorithm for those metric spaces.

Vishwanathan showed how to color 3-colorable graphs on-line with only $O(\sqrt{n \log n})$ colors [Vn], a great improvement over the $O((n \log\log\log n)/\log\log n)$ bound of [LST]. An exponential improvement in the performance of algorithms that traverse certain layered graphs was exhibited by Fiat et al. [FFKRRV].

No superconstant lower bound was known for the competitive ratio of general metric spaces (with at least $k+1$ points). Indeed, several researchers conjectured that constant-competitive randomized k-server algorithms exist for certain infinite metric spaces. No superconstant lower bound for the on-line motion-planning problem was known either.

We prove two k-server lower bounds:

1. The competitive ratio of any randomized k-server algorithm for any sufficiently large metric space is $\Omega(\log k)$.

2. The competitive ratio of any randomized k-server algorithm for *any* metric space with at least $k + 1$ points is $\Omega(\log \log k)$.

In light of the [FKLMSY] algorithm, the first result is tight to within a constant factor.

Blum, Raghavan, and Schieber [BRS] used randomized k-server algorithms to construct randomized on-line motion-planning algorithms. Instead, we adapt randomized k-server lower bounds to prove a lower bound for the on-line motion-planning problem:

3. The competitive ratio of any randomized motion-planning algorithm is $\Omega(\log \log n)$, where n is the number of obstacles in the scene.

The proof of the first result goes as follows. First, we prove that for a *superincreasing* metric space $\mathcal{M}(k)$, a metric space on $\{z_0, z_1, \ldots, z_k\}$ with the distance from z_i to z_{i+1} much greater than that from z_{i-1} to z_i, no sublogarithmic competitive ratio is possible. This part of the proof generalizes techniques of Karlin et al. [KMMO]. Next, we prove a "Ramsey-like" theorem for metric spaces: every sufficiently large metric space $\mathcal{M}$ contains a $(k + 1)$-point subspace resembling either the superincreasing metric space or the uniform metric space. From the $\Omega(\log k)$ lower bounds for the superincreasing and uniform metric spaces we construct an $\Omega(\log k)$ lower bound for $\mathcal{M}$.

For $(k + 1)$-point metric spaces, we will use the Ramsey-like theorem to generate a set $\mathcal{S}$ of $s + 1$ points (where $s = \lceil \sqrt{\lg(k + 1)} \rceil$) that resembles either the superincreasing or uniform metric space. The $k - s$ servers initially on the $k - s$ points outside of $\mathcal{S}$ can be fixed at their initial locations by replacing a request to a point $z \in \mathcal{S}$ by a long sequence of requests to z and all the points outside of $\mathcal{S}$. Thus the on-line algorithm can be forced to use only s servers on the specified $s + 1$ points. We can use the first result to get an $\Omega(\log s)$ bound, and this is $\Omega(\log \log k)$.

The idea for the motion-planning result is to construct an obstacle scene in which the distance moved by an on- or off-line server to reach the target is approximately the cost incurred by a k-server algorithm when serving a sequence of requests in a metric space that resembles $\mathcal{M}(k)$. The lower bound for $\mathcal{M}(k)$ will yield a lower bound on the competitive ratio of motion-planning algorithms.

2. The superincreasing metric space.

Define $c_1 = 1$, and let

$$c_i = c_{i-1} \left[1 + 1/(2e^{2c_{i-1}} - 1) \right]$$

for $i \geq 2$.

DEFINITION. *Let $k \geq 0$, and let $\mathcal{M}(k)$ be a metric space on $k + 1$ points $z_0, z_1, z_2, \ldots, z_k$. Suppose that there are integers $d_1 < d_2 < d_3 < \cdots < d_k$ so that $d_1 = 1$ and that for $i < j$,* $\operatorname{dist}(z_i, z_j) = d_j$. *Then the sequence $d_1, d_2, d_3, \ldots, d_k$ and the metric space $\mathcal{M}(k)$ are called* superincreasing *if $d_i \geq (4c_{i-1}e^{2c_{i-1}})d_{i-1}$ for every $i \geq 2$.*

We will often use $\mathcal{M}(k)$ to mean any superincreasing metric space on $k + 1$ points.

For the metric space $\mathcal{M}(k)$, we prove that $c_k - 1$ is a lower bound on the competitive ratio of any on-line (randomized) k-server algorithm.

The following lemmas show that c_k is $\Theta(\log k)$.

LEMMA 2.1. *$c_k > \frac{1}{2} \ln k$ for all k.*

Proof. The definition of c_i gives the following:

$$c_k - c_{k-1} = \frac{c_{k-1}}{2e^{2c_{k-1}} - 1},$$

or

$$(c_k - c_{k-1})(2e^{2c_{k-1}} - 1) = c_{k-1} \geq 1,$$

or

$$(c_k - c_{k-1})e^{2c_{k-1}} \geq \tfrac{1}{2}.$$

Now, if f is a nondecreasing continuous function and $x_1 < x_2 < \cdots < x_r$, then

$$\int_{x_1}^{x_r} f(x)\,dx \geq \sum_{i=2}^{r} (x_i - x_{i-1}) f(x_{i-1}).$$

Thus

$$\int_{c_1}^{c_k} e^{2x}\,dx \geq \sum_{j=2}^{k} (c_j - c_{j-1}) e^{2c_{j-1}} \geq \frac{k-1}{2}.$$

Therefore,

$$\frac{e^{2c_k} - e^{2c_1}}{2} \geq \frac{k-1}{2},$$

$$e^{2c_k} \geq e^2 + (k-1) > k, \qquad 2c_k > \ln k, \qquad c_k > \tfrac{1}{2}\ln k. \qquad \square$$

LEMMA 2.2. *For all k, $c_k \leq 1 + 1.5 \ln k$.*

Proof. For all y, $1 + y \leq e^y$. Therefore, $1 + y \leq 2e^{2y}$ for all $y \geq 0$.

$$1 + c_{k-1} \leq 2e^{2c_{k-1}},$$

$$c_{k-1} \leq 2e^{2c_{k-1}} - 1.$$

Therefore,

$$\frac{c_{k-1}}{2e^{2c_{k-1}} - 1} \leq 1,$$

$$c_k - c_{k-1} = \frac{c_{k-1}}{2e^{2c_{k-1}} - 1} \leq 1.$$

Therefore, $c_k \leq k$ for all k.

$$\begin{aligned}
(c_k - c_{k-1})2e^{2c_{k-1}} &= c_{k-1} + (c_k - c_{k-1}) = c_k,\\
(c_k - c_{k-1})2e^{2c_k} &\leq (c_k - c_{k-1})2e^{2+2c_{k-1}}\\
&= e^2(c_k - c_{k-1})2e^{2c_{k-1}}\\
&= e^2 c_k \leq ke^2.
\end{aligned}$$

Therefore,

$$(c_k - c_{k-1})2e^{2c_k} \le ke^2,$$

$$(c_k - c_{k-1})e^{2c_k} \le \frac{ke^2}{2}.$$

If f is continuous and nondecreasing and $x_1 < x_2 < \cdots < x_r$, then

$$\int_{x_1}^{x_r} f(x)\,dx \le \sum_{i=2}^{r}(x_i - x_{i-1})f(x_i).$$

Hence

$$\int_{c_1}^{c_k} e^{2x}\,dx \le \sum_{i=2}^{k}(c_i - c_{i-1})e^{2c_i} \le \sum_{i=2}^{k}\frac{ke^2}{2} \le \frac{k^2e^2}{2}.$$

Thus

$$\frac{e^{2c_k} - e^{2c_1}}{2} \le \frac{k^2e^2}{2},$$

so that

$$e^{2c_k} \le e^2 + k^2e^2 = e^2(k^2+1).$$

Hence

$$c_k \le \frac{2 + 3\ln k}{2} = 1 + 1.5\ln k. \qquad \square$$

Let us generalize the k-server problem. The *multipoint k-server problem* is the problem of serving requests each of which consists of a *set* of at most k points. (A request to $\{z\}$ will be abbreviated as z.) To serve the request means to move one or more servers so that *all* the points in the requested set are covered. We will only study multipoint k-server problems on $(k+1)$-point metric spaces. This allows us to assume without loss of generality that the algorithm is *lazy*—it never moves more than one server to serve a request, and if the requested set is already covered, it does nothing.

In $\mathcal{M}(k)$, let τ_i be the set $\{z_1, z_2, z_3, \ldots, z_i\}$. Let γ_i be the set $\{z_0, z_1, z_2, \ldots, z_{i-1}\}$. Let $N_1, N_2, N_3, \ldots$ denote the integer sequence $N_1 = 2$ and $N_i = (2d_i)N_{i-1} + 2$ for $i \ge 2$.

Fix k and a lazy, randomized multipoint k-server algorithm A for $\mathcal{M}(k)$. If $1 \le i \le k$, say a request sequence σ is *i-convergent for A*, or simply *i-convergent*, if A covers τ_i with probability 1 just after σ is served.

LEMMA 2.3. *For each $1 \le i \le k$, for each i-convergent request sequence σ for $\mathcal{M}(k)$, there is a request sequence Δ with the following properties*:

1. *Δ consists of a request sequence for $\mathcal{M}(i-1)$ preceded immediately by a request to z_0 and followed immediately by a request to τ_i. (Thus $\sigma\Delta$ is i-convergent.)*
2. *The length of Δ is at most N_i.*
3. *Suppose A serves request sequence $\sigma\Delta$. Let $s_1, s_2, \ldots, s_i$ be the servers that occupy $\{z_1, z_2, \ldots, z_i\}$ just after σ is served, s_j occupying z_j. Let $t = P[z_0$ is vacant just after A has served $\sigma\Delta]$. Define the i-*cost *of Δ to be the cost incurred*

by $s_1, s_2, \ldots, s_i$ during the time A is serving the Δ of $\sigma\Delta$. Let $w \in \{1, 2, \ldots, 2d_i\}$ be the optimal cost of serving Δ by an i-server algorithm whose servers start on $\{z_1, \ldots, z_i\}$. Then the expected i-cost of Δ is at least $c_i wt$.

Intuitively, the reason why such a costly sequence Δ exists is that the on-line algorithm does not know in advance how many requests in $\mathcal{M}(i-1)$ will be given before the request to τ_i. Suppose that z_0 is vacant prior to the arrival of Δ. If only a few requests in $\mathcal{M}(i-1)$ appear, it makes little sense to move a server on one of the distant points $\{z_i, \ldots, z_k\}$ and pay a high price. If there are many requests, however, a distant server must be moved in order to avoid indefinitely having to shuffle $i-1$ servers among the i points of $\mathcal{M}(i-1)$.

To prove Lemma 2.3, we need a technical lemma, the proof of which appears in the appendix.

LEMMA 2.4. *Let $2 \le i \le k$. Let ℓ be an integer, $\ell \ge c_{i-1}$. Let $d \ge 4c_{i-1}e^{2c_{i-1}}\ell$ be an integer. Let $w_1, w_2, \ldots, w_Q \in \{1, 2, \ldots, 2\ell\}$ where Q satisfies $\sum_{i=1}^{Q-1} w_i < 2d \le \sum_{i=1}^{Q} w_i$. Let $p_1, p_2, \ldots, p_{Q-1} \in [0, 1]$.*

Then either

$$d + c_{i-1} \sum_{s=1}^{Q-1} p_s w_s \ge c_i(2d)$$

or there is an $h \in \{1, 2, \ldots, Q-1\}$ such that

$$d(1-p_h) + c_{i-1} \sum_{s=1}^{h} p_s w_s \ge c_i \sum_{s=1}^{h} w_s.$$

Proof of Lemma 2.3. By induction on i.

Basis: $i = 1$. Let $\Delta = z_0 z_1$. The optimal cost w of serving Δ by a 1-server algorithm whose server starts on z_1 is 2. Algorithm A leaves z_0 vacant after serving $\sigma\Delta$ with probability t, so with probability at least t server s_1 must have served both requests, incurring an expected i-cost of at least $2t = c_i wt$.

Inductive Step: $i > 1$. Let σ be an i-convergent request sequence for $\mathcal{M}(k)$. It is also $(i-1)$-convergent. By induction, there is a request sequence Δ_1 (of length at most N_{i-1}) consisting of a request sequence for $\mathcal{M}(i-2)$ preceded by a request to z_0 and followed by a request to τ_{i-1} such that the following holds. If A serves $\sigma\Delta_1$, the expected $(i-1)$-cost incurred while serving Δ_1 is at least $c_{i-1}w_1t_1$ (where $w_1 \in \{1, 2, \ldots, 2d_{i-1}\}$ is the optimal cost of serving Δ_1 with $i-1$ servers and t_1 is the probability that z_0 is vacant after $\sigma\Delta_1$ is served).

Since $\sigma\Delta_1$ is $(i-1)$-convergent, we can apply induction again. Thus there is a Δ_2 (of length at most N_{i-1}) consisting of a request sequence for $\mathcal{M}(i-2)$ preceded by a request to z_0 and followed by a request to τ_{i-1} such that the following holds. If A serves $\sigma\Delta_1\Delta_2$, the expected $(i-1)$-cost incurred while serving Δ_2 is at least $c_{i-1}w_2t_2$ (where $w_2 \in \{1, 2, \ldots, 2d_{i-1}\}$ is the optimal cost of serving Δ_2 with $i-1$ servers and t_2 is the probability that z_0 is vacant after $\sigma\Delta_1\Delta_2$ is served).

Since $\sigma\Delta_2$ is $(i-1)$-convergent, we can apply induction again. Thus there is a Δ_3 (of length at most N_{i-1}) consisting of a request sequence for $\mathcal{M}(i-2)$ preceded by a request to z_0 and followed by a request to τ_{i-1} such that the following holds. If A serves $\sigma\Delta_1\Delta_2\Delta_3$, the expected $(i-1)$-cost incurred while serving Δ_3 is at least $c_{i-1}w_3t_3$ (where $w_3 \in \{1, 2, \ldots, 2d_{i-1}\}$ is the optimal cost of serving Δ_3 with $i-1$ servers and t_3 is the probability that z_0 is vacant after $\sigma\Delta_1\Delta_2\Delta_3$ is served).

Repeat this process, getting $\Delta_1, \Delta_2, \ldots, \Delta_Q$ and $w_1, w_2, \ldots, w_Q$ such that $w_1 + w_2 + \cdots + w_Q \geq 2d_i$ but $w_1 + \cdots + w_{Q-1} < 2d_i$. Now change Δ_Q—replace it by $\Delta_Q \gamma_i$.

Let "time j" mean "just after $\sigma\Delta_1 \cdots \Delta_j$ has been served." Let $t_j = P[z_0$ is vacant at time $j]$. Let $u_j = P[z_i$ is vacant at time $j]$, and let $r_j = t_j + u_j$, the probability that either z_0 or z_i is vacant at time j. Because $r_j = P[z_{i+1}, z_{i+2}, \ldots, z_k$ are occupied at time $j]$ (even for $j = Q$), we have $r_1 \geq r_2 \geq \cdots \geq r_Q$. Let

$$q_j = \begin{cases} u_j/r_j & \text{if } r_j \neq 0, \\ 1 & \text{otherwise.} \end{cases}$$

Note that $r_j q_j = u_j$ always. The expected $(i-1)$-cost of phase $j < Q$ is at least

$$c_{i-1} w_j t_j = c_{i-1} w_j (r_j - u_j) = c_{i-1} w_j r_j (1 - q_j).$$

Clearly $t_Q = 0$. Therefore, $r_Q = u_Q$, so $q_Q = 1$. (Of course, the expected $(i-1)$-cost of phase Q is at least $0 = c_{i-1} w_Q r_Q (1 - q_Q)$.) $\Delta_1\Delta_2 \cdots \Delta_Q$ is a request sequence for $\mathcal{M}(i-1)$ preceded by a request to z_0.

Choose $1 \leq h \leq Q$. The expected cost incurred by s_i to serve $\Delta_1\Delta_2 \cdots \Delta_h \tau_i$ (after A has already served σ) is at least $d_i u_h$, since z_i is occupied just after σ is served. Thus the expected i-cost, if we have $h \leq Q$ phases, is at least

$$\begin{aligned} d_i u_h + \textstyle\sum_{j=1}^{h} c_{i-1} w_j r_j (1 - q_j) &\geq r_h (d_i q_h) + c_{i-1} r_h \textstyle\sum_{j=1}^{h} w_j (1 - q_j) \\ &= r_h \left(d_i q_h + c_{i-1} \textstyle\sum_{j=1}^{h} w_j (1 - q_j)\right) \end{aligned}$$

and $q_Q = 1$. When $h = Q$, this last quantity is

$$r_Q \left(d_i + c_{i-1} \sum_{j=1}^{Q-1} w_j (1 - q_j)\right).$$

By Lemma 2.4, with $\ell = d_{i-1}$, $d = d_i$, and $p_s = 1 - q_s$ for all s, either

$$d_i + c_{i-1} \sum_{j=1}^{Q-1} w_j (1 - q_j) \geq c_i (2d_i)$$

or there is an $h \in \{1, 2, \ldots, Q-1\}$ such that

$$d_i q_h + c_{i-1} \sum_{j=1}^{h} w_j (1 - q_j) \geq c_i \sum_{j=1}^{h} w_j.$$

In the former case, the optimal cost incurred by an adversary having i servers in serving $\Delta_1\Delta_2 \cdots \Delta_Q \tau_i$ after serving σ equals $2d_i$. A's expected i-cost to serve the same sequence after serving σ is at least $r_Q[c_i(2d_i)]$. Let $\Delta = \Delta_1\Delta_2 \cdots \Delta_Q \tau_i$. If z_0 is vacant after $\sigma\Delta$ is served, then either z_0 or z_i is vacant after $\sigma\Delta_1\Delta_2 \cdots \Delta_Q$ is served. Thus the probability t that z_0 is vacant after $\sigma\Delta$ is served is at most r_Q, and therefore Δ suffices, since Δ's length is at most $(2d_i)N_{i-1} + 2 = N_i$. ($Q \leq 2d_i$ since $w_j \geq 1$ for all j.)

In the latter case, the optimal cost incurred by an i-server adversary in serving $\Delta_1\Delta_2 \cdots \Delta_h \tau_i$ after serving σ is at most $\sum_{j=1}^{h} w_j$. A's expected i-cost to serve the same sequence after serving σ is at least $r_h c_i \sum_{j=1}^{h} w_j$. Let $\Delta = \Delta_1\Delta_2 \cdots \Delta_h \tau_i$. If z_0 is vacant after $\sigma\Delta$ is served, then either z_0 or z_i is vacant after $\sigma\Delta_1\Delta_2\cdots\Delta_h$ is served. Thus the probability t that z_0 is vacant after $\sigma\Delta$ is served is at most r_h, and therefore Δ suffices. □

We use the notation $\mathrm{OPT}(\sigma)$ to denote the optimal off-line multipoint cost to serve σ, and we use $A(\sigma)$ to denote the (random) cost of our on-line multipoint algorithm A to serve σ.

THEOREM 2.5. *For all r, there is a multipoint request sequence σ_r of length at most rN_k and optimal cost at least r such that*

$$E[A(\sigma_r)] \geq c_k \cdot \mathrm{OPT}(\sigma_r).$$

Proof. Take $i = k$. Build request sequence $\sigma_r = \Delta_1\Delta_2\Delta_3\cdots\Delta_r$ via repeated applications of Lemma 2.3, by constructing Δ_1, then Δ_2, then Δ_3, and so on. Each Δ_j has length at most N_k. The optimal cost of serving Δ_j with k servers is $w_j \geq 1$. Each time the lemma is applied, $t = 1$.

$$\mathrm{OPT}(\sigma_r) = w_1 + w_2 + \cdots + w_r.$$

(We have equality because each Δ_j ends with τ_k.) The expected value of $A(\sigma_r)$ is at least $c_k w_1 + c_k w_2 + \cdots + c_k w_r = c_k \cdot \mathrm{OPT}(\sigma_r)$. □

COROLLARY 2.6. *There is no c-competitive multipoint k-server algorithm for $\mathcal{M}(k)$ if $c < c_k$.*

DEFINITION. *Let $\mathcal{M}$ be a metric space on the $k+1$ points $\{z_0, z_1, \ldots, z_k\}$. Let $S \subsetneq \mathcal{M}$, $S \neq \emptyset$. Say an S-*request *is a sequence of requests to all $|S|$ points in S, one at a time, in increasing order by index.*

DEFINITION. *Let $\mathcal{M}$ be a metric space on the $k+1$ points $\{z_0, z_1, \ldots, z_k\}$. We say a (single-point) server algorithm A for $\mathcal{M}$ is finitely converging if it has the following property. Let α be a request sequence, and let $S \subsetneq \mathcal{M}$, $S \neq \emptyset$. If A serves a sequence consisting of α followed by enough S-requests, then at the end all the points in S are occupied with probability one.*

LEMMA 2.7. *Let $\mathcal{M}$ be a metric space on $k+1$ points. If there is a lazy, c-competitive, finitely converging algorithm for the single-point k-server problem on $\mathcal{M}$, then there is a lazy, c-competitive multipoint k-server algorithm for $\mathcal{M}$.*

Proof. Suppose A is a lazy, finitely converging c-competitive k-server algorithm for $\mathcal{M}$. We argue that there is a lazy, c-competitive multipoint k-server algorithm B for $\mathcal{M}$. Let σ be a multipoint request sequence for $\mathcal{M}$. B simulates A on a single-point request sequence σ' for $\mathcal{M}$. B constructs σ' on the fly by replacing a request to the set S in σ by a long sequence of S-requests. The.number of these S-requests is chosen so large that A is known to cover S with probability one after serving the S-requests. Further, the number is chosen so large that even the adversary is known to cover S afterward.

To serve a request to S, B flips coins for A and "watches" A's behavior on the long string of S-requests. At the end, B moves to A's configuration by moving at most one server.

Let OPT and OPT′ denote the optimal costs of a multipoint and single-point request sequence, respectively. $B(\sigma) \leq A(\sigma')$ and hence

$$E[B(\sigma)] \leq E[A(\sigma')] \leq c \cdot \mathrm{OPT}'(\sigma') + a$$

for a suitable a. But $\mathrm{OPT}(\sigma) = \mathrm{OPT}'(\sigma')$, and thus B is a lazy, c-competitive, multipoint k-server algorithm for $\mathcal{M}$. □

THEOREM 2.8. *There is no lazy, finitely converging c-competitive algorithm for the single-point k-server problem on $\mathcal{M}(k)$ if $c < c_k$.*

Proof. Follows from Corollary 2.6 and Lemma 2.7. □

Now we relate finitely converging and nonfinitely converging algorithms.

LEMMA 2.9. *Suppose that A is a lazy, c-competitive k-server algorithm for a $(k+1)$-point metric space M. Then there is a lazy, finitely converging $(c+1)$-competitive k-server algorithm A' for M.*

Proof sketch. Let the points of M be ordered $z_0, z_1, \ldots, z_k$. Without loss of generality, suppose that the minimum nonzero distance in M equals one and the maximum distance equals, say, D. Choose $a \geq 0$ such that $E[A(\sigma)] \leq c \cdot \mathrm{OPT}(\sigma) + a$ for all σ.

At all times, A' simulates either A or the deterministic, lazy, k-competitive algorithm BAL of [MMS], initially the former. A long sequence of S-requests will ensure that A fails to occupy all the points of S with probability approaching zero, for otherwise it could not be competitive. (Indeed, enough S-requests will ensure that any given competitive algorithm, on-line or off-line, occupies S with probability approaching one. The adversary himself can be forced to occupy S in this way.)

At all times, A' attempts to write the list of requests seen to date, including the current request, as τS^L, where

$$L = L(\tau) = 1 + \lceil k^2 D|\tau|^2(c|\tau| + 2cD + a)\rceil.$$

(Here, S represents an S-request.) If it fails, it simply flips coins and serves the request as A would have. If it succeeds—and in this case τ and S are unique—A' flips its coins to simulate A on the current request. If the coin flips dictate that A move to a configuration covering S, A' continues to mimic A. Otherwise, A' switches to BAL in a lazy way, never to return to A. If the next $|S|$ requests are an S-request, then L is so large that after those requests are served A' covers S. Thus A' covers S with probability one by the time it has served τS^{L+1}.

It is not hard to prove that L is so large that the probability that A' switches to BAL when the list of requests seen to date is of the form τS^L is at most $1/(k^2 D|\tau|^2)$. Therefore, the probability that A' ever switches to BAL is at most $\sum_{l=1}^{\infty} 1/(k^2 D l^2) \leq 2/(k^2 D)$. Now $A'(\sigma) = A(\sigma)$ if A' never switches to BAL, and $A'(\sigma) \leq A(\sigma) + (D + \mathrm{BAL}(\sigma))$ if A' does switch.

$$\begin{aligned} D + \mathrm{BAL}(\sigma) &\leq D + \left(k \cdot \mathrm{OPT}(\sigma) + \binom{k}{2} D\right) \\ &\leq k \cdot \mathrm{OPT}(\sigma) + (k^2/2)D. \end{aligned}$$

Therefore,

$$\begin{aligned} E[A'(\sigma)] &\leq E[A(\sigma)] + (k \cdot \mathrm{OPT}(\sigma) + \frac{k^2}{2} D) \cdot P[A' \text{ switches to BAL}] \\ &\leq E[A(\sigma)] + \frac{2}{kD} \cdot \mathrm{OPT}(\sigma) + 1 \\ &\leq E[A(\sigma)] + (\mathrm{OPT}(\sigma) + 1) \\ &\leq (c+1) \cdot \mathrm{OPT}(\sigma) + (a+1). \qquad \Box \end{aligned}$$

THEOREM 2.10. *There is no c-competitive algorithm A for the single-point k-server problem on $\mathcal{M}(k)$ if $c < c_k - 1$.*

Proof. Lemma 2.9 proves that the existence of a c-competitive algorithm implies the existence of a lazy, finitely converging $(c+1)$-competitive algorithm. Theorem 2.8 completes the proof. □

3. Ramsey theory for metric spaces. This section deals with the structure of metric spaces. Theorem 3.2 shows that every metric space of cardinality n contains either a roughly uniform subset of at least $\frac{1}{2}\lg n$ points or an approximately superincreasing sequence of $\Omega((\lg n)/\lg\lg n)$ points.

LEMMA 3.1. *Let $\mathcal{M}$ be a metric space on n points, $w(e)$ denoting the length of edge e, and let $c > 1$ be a real. $\mathcal{M}$ contains either*:

1. *a subset S of size at least $s = \lg n$ such that the distances within S differ by no more than a factor of c^2 or*

2. *a point P and a subset T of size at least $t = n/\lg n$ such that $P \notin T$ and*

$$\frac{\min_{x\in T} d(x, P)}{\max_{x,y\in T} d(x, y)} \geq \frac{c}{4} - \frac{1}{2}.$$

Proof. We may assume $n \geq 3$. Label edge e with $\lfloor \log_c w(e) \rfloor$. Let j be the largest label, and call those edges labeled j or $j-1$ *large* and the rest, if any, *small*. Build a graph G on the points of $\mathcal{M}$: $\{u, v\} \in E(G)$ if and only if $\{u, v\}$ is small in $\mathcal{M}$. The value of d, a nonnegative real, will be chosen later.

If every vertex in G has degree at most d, then G has an independent set S of size at least $n/(d+1)$. (The greedy independent set algorithm generates such an S.) If u and v are distinct points of S, $\{u, v\}$ is large. But the lengths of two large edges cannot differ by more than a factor of c^2.

Otherwise, some vertex v has degree exceeding d. Let $\{a, b\}$ be the longest edge in the metric space, labeled so that $d(a, v) \geq d(b, v)$; $d(a, v) \geq \frac{1}{2}d(a, b) \geq \frac{1}{2}c^j$. Let T be v together with its neighborhood in G. All edges between points in T are of length less than $2c^{j-1}$. If $x \in T$,

$$d(a, x) \geq d(a, v) - d(v, x) \geq \tfrac{1}{2}c^j - c^{j-1}.$$

Thus

$$\frac{\min_{x\in T} d(x, a)}{\max_{x,y\in T} d(x, y)} \geq \frac{c^j/2 - c^{j-1}}{2c^{j-1}} = \frac{c}{4} - \frac{1}{2}.$$

We have found either a set S of size at least $n/(d+1)$ whose interpoint distances differ by at most a factor of c^2 or a set T for which

$$\frac{\min_{x\in T} d(x, a)}{\max_{x,y\in T} d(x, y)} \geq \frac{c}{4} - \frac{1}{2}$$

with $|T| > 1 + d$. Taking $d = (n/\lg n) - 1$ and $P = a$, the proof is complete. □

THEOREM 3.2. *Let $\mathcal{M}$ be a metric space on n points, and let $c > 1$. $\mathcal{M}$ contains either*

1. *a subset of size at least $\frac{1}{2}\lg n$ whose interpoint distances differ by at most a factor of c^2 or*

2. *a sequence of distinct vertices $P_1, P_2, \ldots, P_t$ for $t = \lceil \frac{1}{2}(\lg n)/\lg\lg n \rceil - 1$ such that*

$$\frac{\min_{j>i} d(P_j, P_i)}{\max_{j>i+1} d(P_j, P_{i+1})} \geq \frac{c}{4} - \frac{1}{2}$$

for $i = 1, 2, \ldots, t-2$.

Proof. Construct a sequence of metric spaces $\mathcal{M} = \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, \ldots, \mathcal{M}_r$ and a set of points $P_1, P_2, \ldots, P_{r-1}$ (P_i in $\mathcal{M}_i$) as follows. Apply Lemma 3.1 to $\mathcal{M}_i$. If case 1 is true (where $n = |\mathcal{M}|$, not $|\mathcal{M}_i|$), halt. If case 2 is true but T has fewer than $\sqrt{n}$ points, halt without constructing $\mathcal{M}_{i+1}$. Otherwise, let P_i be the point P of case 2 and $\mathcal{M}_{i+1}$ be the metric space induced by T.

If case 1 is ever true, we have a set of at least $\lg \sqrt{n} = \frac{1}{2} \lg n$ points within the current metric space whose interpoint distances differ by at most c^2. Otherwise, since

$$|\mathcal{M}_{i+1}| \geq \frac{|\mathcal{M}_i|}{\lg |\mathcal{M}_i|} \geq \frac{|\mathcal{M}_i|}{\lg n},$$

the number r of metric spaces we construct satisfies $n/(\lg n)^r < \sqrt{n}$, i.e., $r > \frac{1}{2}(\lg n)/\lg\lg n$. The $r-1$ P's satisfy condition 2 above. □

4. General lower bounds.

LEMMA 4.1. *Let $\mathcal{M}$ and $\mathcal{M}'$ be two metric spaces defined on the same set of points with distance functions d and d', respectively. Let $b \geq 1$. Suppose that for every two points x, y, $d'(x, y) \leq d(x, y) \leq b \cdot d'(x, y)$. Let c be a lower bound on the competitive ratio for $\mathcal{M}'$. Then c/b is a lower bound on the competitive ratio for $\mathcal{M}$.*

Proof. The cost of serving σ in $\mathcal{M}$ is bounded above and below by b times and one times the cost of serving it in $\mathcal{M}'$, respectively. □

LEMMA 4.2. *Let $b \geq 1$, and let $\mathcal{M}$ be a metric space where*

$$\frac{\max_{x,y\in\mathcal{M}} \operatorname{dist}(x, y)}{\min_{x\neq y\in\mathcal{M}} \operatorname{dist}(x, y)} \leq b.$$

Then the competitive ratio for any randomized on-line k-server algorithm for $\mathcal{M}$ is at least $\mathcal{H}_k/b$, where $\mathcal{H}_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$ is the kth harmonic number.

Proof. Scale the distances in $\mathcal{M}$ so that the minimum nonzero distance is 1. Apply Lemma 4.1 and the lower bound of $\mathcal{H}_k$ for a uniform metric space [FKLMSY]. □

LEMMA 4.3. *Let $\mathcal{M}$ be a metric space defined on the $k + 1$ points $\{x_0, \ldots, x_k\}$ by the distance function d, which satisfies*:

1. $d(x_0, x_1) = 1$ *and*
2. *for every i, $1 < i \leq k$,*

$$\frac{\min_{j<i} d(x_j, x_i)}{\max_{j<i-1} d(x_j, x_{i-1})} \geq 8c_{i-1}e^{2c_{i-1}}.$$

Then $(c_k - 1)/4$ is a lower bound on the competitive ratio of any on-line randomized k-server algorithm serving requests in $\mathcal{M}$.

Proof. For all i, $1 \leq i \leq k$, define $a_i = \lceil \min_{0\leq j<i} d(x_j, x_i)\rceil$. Conditions 1 and 2 imply that the sequence $a_1, a_2, \ldots$ is superincreasing. Let $\mathcal{M}'$ be the superincreasing metric space defined on $\{x_0, \ldots, x_k\}$ by setting $\operatorname{dist}(x_j, x_i) = a_i$ for $j < i$. It is not hard to prove that $\frac{1}{2}a_i \leq \operatorname{dist}(x_j, x_i) \leq a_1 + a_2 + \cdots + a_i \leq 2a_i$ for all $j < i$. Theorem 2.10 and Lemma 4.1 complete the proof. □

LEMMA 4.4. *Let $t > 2$, and let $P_0, P_1, \ldots, P_{t-1}$ be a sequence of points in a metric space such that* $\operatorname{dist}(P_0, P_1) = 1$ *and for every i, $1 < i \leq t - 1$,*

$$\frac{\min_{j<i} \operatorname{dist}(P_j, P_i)}{\max_{j<i-1} \operatorname{dist}(P_j, P_{i-1})} \geq 2.$$

If $r > 1$, then there is a subset $\{Q_0, Q_1, \dots, Q_{u-1}\}$ of size $u \geq t/(1 + \lg r)$ so that for every i, $1 < i \leq u$,

$$\frac{\min_{j<i} \operatorname{dist}(Q_j, Q_i)}{\max_{j<i-1} \operatorname{dist}(Q_j, Q_{i-1})} \geq r.$$

Proof sketch. Take every $\lceil \lg r \rceil$th point of the P's. □

THEOREM 4.5. *If $n = |\mathcal{M}|$ is sufficiently large, then there is a lower bound of $\Omega(\log k)$ on the competitive ratio of any randomized on-line k-server algorithm for $\mathcal{M}$.*

Proof. Apply Theorem 3.2 to $\mathcal{M}$ with $c = 10$. If case 1 holds and $\frac{1}{2} \lg n \geq k+1$, we apply Lemma 4.2 to obtain a lower bound of $\mathcal{H}_k/100$. So suppose case 2 holds. Lemma 2.2 implies that $c_i \leq 1 + 1.5 \ln i$, so that $8c_i e^{2c_i} \leq 8(1 + 1.5 \ln i)e^2 i^3$. Define $r = 8(1 + 1.5 \ln k)e^2 k^3$. If

$$\frac{\lceil \frac{1}{2}(\lg n)/\lg\lg n \rceil - 1}{1 + \lg r} \geq k + 1,$$

then Theorem 3.2 and Lemmas 4.3 and 4.4 give us a lower bound of $\frac{1}{4}(c_k - 1) \geq \frac{1}{8} \ln k - \frac{1}{4}$. It is clear that there is a polynomial $p(k)$ so that if $n \geq 2^{p(k)}$, then

$$\frac{\lceil \frac{1}{2}(\lg n)/\lg\lg n \rceil - 1}{1 + \lg r} \geq k + 1. \qquad \square$$

THEOREM 4.6. *For any metric space with at least $k + 1$ points, there is a lower bound of $\Omega(\log\log k)$ on the competitive ratio of every randomized on-line k-server algorithm.*

Proof. Let $\mathcal{M}$ be a metric space on exactly $k + 1$ points. (Ignore any others.) Let $s = \lceil \sqrt{\lg(k + 1)} \rceil$. The technique of Theorem 4.5 can be used to construct an $(s + 1)$-point metric space $\mathcal{S}$ within $\mathcal{M}$ whose interpoint distances either differ by at most a factor of 100 or which "grow" by at least a factor of s^4. (This holds for sufficiently large k.) In either case we have a lower bound of $f(s)$ on the competitive ratio of any randomized s-server algorithm for $\mathcal{S}$, where $f(s)$ is $\Omega(\log s)$ and hence $\Omega(\log\log k)$. However, the algorithm has k servers, not s.

Suppose there is a lazy, finitely converging c-competitive k-server algorithm for $\mathcal{M}$. Then by Lemma 2.7 we infer that there is a lazy, c-competitive multipoint k-server algorithm A for $\mathcal{M}$. Then there is a c-competitive s-server algorithm A' for $\mathcal{S}$: A' simply replaces a request to $z \in \mathcal{S}$ by a request to the set $\{z\} \cup (\mathcal{M} - \mathcal{S})$ and feeds the request to A. Neither A nor the adversary will ever move any server initially outside of $\mathcal{S}$. It follows easily that A' is a c-competitive s-server algorithm for $\mathcal{S}$. However, for $c < f(s)$, this cannot be. Now Lemma 2.9 tells us that no $(f(s) - 1)$-competitive k-server algorithm for $\mathcal{M}$ can exist, finitely converging or not. □

5. On-line motion planning. In this section we study the on-line motion-planning problem of Papadimitriou and Yannakakis, as described in the introduction. [PY] proved that no deterministic search strategy achieves a constant ratio. In this section we prove that even a randomized searcher cannot achieve a constant ratio. Rather, the ratio must grow with the number of obstacles.

Let $d_1, d_2, d_3, \dots, d_k$ be a superincreasing sequence of integers. Set $d_0 = 0$. Define a new metric space $\mathcal{M}'(k)$ on $\{z_0, z_1, \dots, z_k\}$ by identifying z_i with the point d_i on the real line as $\operatorname{dist}(z_i, z_j) = |d_i - d_j|$.

THEOREM 5.1. *For all multipoint k-server algorithms A for $\mathcal{M}'(k)$, for all r, there is a request sequence σ_r of optimal cost at least r and of length at most rN_k such that*

$$E[A(\sigma_r)] \geq \frac{c_k}{2} \cdot \mathrm{OPT}(\sigma_r).$$

Proof. Let A be a multipoint k-server algorithm for $\mathcal{M}'(k)$. Let A' be the algorithm for $\mathcal{M}(k)$ that mimics the behavior of A on $\mathcal{M}'(k)$. Each distance in $\mathcal{M}'(k)$ is between one half and one times the corresponding distance in $\mathcal{M}(k)$. It follows that

$$E[A(\sigma)] \geq \tfrac{1}{2} E[A'(\sigma)]$$

for all σ. By Theorem 2.5, for each r there is a request sequence σ_r' for $\mathcal{M}(k)$ of optimal cost at least r and of length at most rN_k such that

$$E[A'(\sigma_r)] \geq c_k \cdot \mathrm{OPT}(\sigma_r').$$

The optimal cost of σ_r' in $\mathcal{M}(k)$ is at least its optimal cost in $\mathcal{M}'(k)$. We may set $\sigma_r = \sigma_r'$. □

Let $N = N_k$. Fix a randomized searching algorithm. For scenes with at most $(k+2)N+2$ obstacles, we will prove a lower bound of $\Omega(\log k)$ on the performance ratio.

Choose k. Let $h = N + 1$. We first build a collection of $(k + 2)N + 2$ open rectangles as follows. For each $i = 1, 2, \ldots, N$, place k rectangles of width one in the region $\{(x, y) | i-1 \leq x < i\}$ known as *column* i. The jth rectangle C_{ij} $(1 \leq j \leq k)$ runs from $y = d_{j-1}h$ to $y = d_j h$. These k rectangles cover the region $\{(x, y) | i-1 < x < i, 0 < y < hd_k\}$. Now add a rectangle C_{i0} of width one and infinite height just below C_{i1} and add a rectangle $C_{i,k+1}$ of the same size as C_{i0} just above C_{ik}. Now add one infinite open rectangle L covering all of the plane to the left of these $(k+2)N$ rectangles. To their right add an infinite rectangle R covering everything to the right. Now we need to shift the rectangles slightly. Define $\epsilon_i = 2^{-i}$ and slide upward by ϵ_i all $k + 2$ rectangles in column i, $1 \leq i \leq N$.

To summarize, the final positions of the rectangles are as follows: For $1 \leq j \leq k$,

$C_{ij} = \{(x, y) | i - 1 < x < i, d_{j-1}h + \epsilon_i < y < d_j h + \epsilon_i\}$,
$C_{i0} = \{(x, y) | i - 1 < x < i, y < \epsilon_i\}$, and
$C_{i,k+1} = \{(x, y) | i - 1 < x < i, y > hd_k + \epsilon_i\}$.

These rectangles together with L and R cover all but a set of measure 0 of the entire plane.

The input consists of a sequence $\sigma_1, \sigma_2, \ldots, \sigma_N$, chosen in advance, with $\sigma_i \subsetneq \{0, 1, \ldots, k\}$ and $\sigma_i \neq \emptyset$ for all i. In column i, $1 \leq i \leq N$, the adversary "fuses" rectangles C_{ij} and $C_{i,j+1}$ for all $j \in \sigma_i$. The searcher's origin s is $(0, \epsilon_1)$, and his target is the vertical line $x = N$.

As soon as the searcher first reaches column i, i.e., his x-coordinate first reaches $i - 1$, the adversary tells him σ_i. The slight vertical displacement between columns $i - 1$ and i prevents the searcher from learning anything about σ_i before he enters column i.

Against these possible inputs, we can convert any searching algorithm to an algorithm A of no greater cost with this property:

- As soon as the searcher's x-coordinate reaches $i - 1$ (and he learns σ_i), he chooses a column-i rectangle C_{ij} with $0 \leq j \leq k$ and $j \notin \sigma(i)$. (The choice of $j \in \{0, 1, \ldots, k\}$ is random and probably not uniform.) Because $j \notin \sigma(i)$, rectangle C_{ij} has not been fused with its neighbor above. The searcher then moves vertically to the upper-left corner of C_{ij} and then one unit rightward.

Let $a_i \notin \sigma_i$ be the searcher's random choice of j in column i, and let $a_0 = 0$. His cost of moving from the upper-right corner $(i-1, d_{a_{i-1}}h + \epsilon_{i-1})$ of $C_{i-1,a_{i-1}}$ to the upper-right corner $(i, d_{a_i}h + \epsilon_i)$ of C_{i,a_i} is at least $h|d_{a_i} - d_{a_{i-1}}|$. His total cost is therefore at least $h\sum_{i=1}^{N}|d_{a_i} - d_{a_{i-1}}|$. The random choice of $a_i \notin \sigma_i$ depends only on $\sigma_1, \sigma_2, \ldots, \sigma_i$. This is precisely the situation of a randomized multipoint k-server algorithm that serves N multipoint requests in $\mathcal{M}'(k)$ and starts with its "hole" at d_0: d_{a_i} is the location of the algorithm's "hole" after serving the ith request σ_i. We will view A as a multipoint k-server algorithm for $\mathcal{M}'(k)$.

Given $\sigma_1, \ldots, \sigma_N$, set $b_0 = 0$ and let $b_1, b_2, \ldots, b_N$ be the optimal way to serve requests $\sigma_1, \ldots, \sigma_N$ in $\mathcal{M}'_k$. In other words, $b_1, b_2, \ldots, b_N$ minimizes $\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|$ subject to the constraint $b_i \notin \sigma_i$ for all i. Then the length of the shortest obstacle-free $s - t$ path is at most

$$\sum_{i=1}^{N}[1 + 2^{-i} + h|d_{b_i} - d_{b_{i-1}}|] < N + 1 + h\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|.$$

By Theorem 5.1 applied to A with $r = 1$, there is a request sequence σ of length at most N such that

$$E[A(\sigma)] \geq \frac{c_k}{2} \cdot \mathrm{OPT}(\sigma).$$

(Theorem 5.1 goes through even if A knows the length of the request sequence in advance.) For that sequence,

$$\frac{E[\sum_{i=1}^{N}|d_{a_i} - d_{a_{i-1}}|]}{\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|} \geq \frac{c_k}{2}.$$

Thus

$$\begin{aligned}\frac{E[h\sum_{i=1}^{N}|d_{a_i} - d_{a_{i-1}}|]}{N+1+h(\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|)} &= \frac{hE[\sum_{i=1}^{N}|d_{a_i} - d_{a_{i-1}}|]}{h(1+\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|)}\\ &\geq \frac{E[\sum_{i=1}^{N}|d_{a_i} - d_{a_{i-1}}|]}{2\sum_{i=1}^{N}|d_{b_i} - d_{b_{i-1}}|}\\ &= \frac{c_k}{4}.\end{aligned}$$

With at most $2 + (k+2)N$ rectangles, we have proven a lower bound of $c_k/4$, which is $\Omega(\log k)$. If desired, each of the rectangles with an infinite side length can be replaced by a rectangle with finite but very large dimensions.

We choose

$$d_k = \lceil 4c_{k-1}e^{2c_{k-1}} \rceil d_{k-1}$$

for all $k \geq 2$ and $d_1 = 1$. A simple calculation yields $N_k \leq 4^k(d_1 d_2 \cdots d_k)$, $d_k \leq k^{10}d_{k-1}$, and $N_k \leq 4^k(k!)^{10k}$. This means that we have a lower bound of $(\ln\ln n)/24$, n being the number of rectangles, for infinitely many n.

6. Acknowledgments. We are grateful to Prabhakar Raghavan for inspiring us to seek a general k-server lower bound, to Lyle McGeoch for his help in fixing a mistake, to Mario

Szegedy for simplifying the proof of Lemma 3.1, and to Dean Foster and Amos Fiat for their helpful remarks.

7. Appendix. Let $\beta \geq 1$ be a real, and let $\ell \geq \beta$ be a positive integer. Let $d \geq 4\beta e^{2\beta}\ell$ be an integer. Let $w_1, w_2, \ldots, w_Q \in \{1, 2, \ldots, 2\ell\}$ where Q satisfies $\sum_{i=1}^{Q-1} w_i < 2d \leq \sum_{i=1}^{Q} w_i$.

We prove a lower bound on the solution of the following linear program LP$_1$:

Find $p_1, p_2, \ldots, p_{Q-1}, \gamma$ so as to minimize γ subject to

$$d(1 - p_t) + \beta \cdot \sum_{s=1}^{t} p_s w_s \leq \gamma \cdot \sum_{s=1}^{t} w_s \quad \text{for } t = 1, \ldots, Q-1$$

$$d + \beta \cdot \sum_{s=1}^{Q-1} p_s w_s \leq \gamma \cdot 2d,$$

and

$$0 \leq p_s \leq 1 \quad \text{for } s = 1, \ldots, Q-1.$$

We first prove that the solution of the linear program LP$_2$:

Find $r_1, r_2, \ldots, r_{2d-2\ell}, \gamma$ so as to minimize γ subject to

$$d(1 - r_j) + \beta \cdot \sum_{i=1}^{j} r_i \leq \gamma \cdot (j + 2\ell) \quad \text{for } j = 1, \ldots, 2d - 2\ell$$

and

$$d + \beta \cdot \sum_{i=1}^{2d-2\ell} r_i \leq \gamma \cdot 2d$$

is a lower bound on the solution of LP$_1$. Then we prove that the solution of the linear program LP$_3$:

Find $t_1, t_2, \ldots, t_{2d-2\ell}, \gamma$ so as to minimize γ subject to

$$d(1 - t_j) + \beta \cdot \sum_{i=1}^{j} t_i = \gamma \cdot (j + 2\ell) \quad \text{for } j = 1, \ldots, 2d - 2\ell$$

and

$$d + \beta \cdot \sum_{i=1}^{2d-2\ell} t_i \leq \gamma \cdot 2d$$

is a lower bound on the solution of LP$_2$. LP$_2$ and LP$_3$ are identical, except that the first $2d - 2\ell$ inequalities in LP$_2$ are equalities in LP$_3$.

LEMMA 7.1. *The solution of* LP$_2$ *is a lower bound on the solution of* LP$_1$.

Proof. Suppose $p_1, \ldots, p_{Q-1}, \gamma$ is a feasible solution to LP$_1$. Then the assignment

$$\begin{aligned} r_1 &= p_1 \\ r_2 &= p_1 \\ &\vdots \end{aligned}$$

$$\begin{aligned} r_{w_1} &= p_1 \\ r_{w_1+1} &= p_2 \\ r_{w_1+2} &= p_2 \\ &\vdots \\ r_{w_1+w_2} &= p_2 \\ &\vdots \\ r_{w_1+\cdots+w_{Q-2}+1} &= p_{Q-1} \\ r_{w_1+\cdots+w_{Q-2}+2} &= p_{Q-1} \\ &\vdots \\ r_{w_1+\cdots+w_{Q-1}} &= p_{Q-1} \end{aligned}$$

and γ is a solution to LP$_2$. (Because $w_1 + w_2 + \cdots + w_{Q-1} \geq 2d - 2\ell$, we have constructed at least as many r's as we need, maybe more.) □

LEMMA 7.2. *The solution of* LP$_3$ *is a lower bound on the solution of* LP$_2$.

Proof. Suppose $r_1, r_2, \ldots, r_{2d-2\ell}, \gamma$ is a feasible solution to LP$_2$. Define $R_0 = 0$ and $R_j = r_1 + r_2 + \cdots + r_j$, $1 \leq j \leq 2d - 2\ell$. Then

$$d(1 - R_j + R_{j-1}) + \beta R_j \leq \gamma(j + 2\ell)$$

for $1 \leq j \leq 2d - 2\ell$ and

$$d + \beta R_{2d-2\ell} \leq \gamma \cdot 2d.$$

Therefore,

$$d - R_j(d - \beta) + dR_{j-1} \leq \gamma(j + 2\ell),$$

which is equivalent to

$$R_j \geq \frac{d}{d - \beta} R_{j-1} + \frac{d - \gamma(j + 2\ell)}{d - \beta}.$$

Also $R_{2d-2\ell} \leq d(2\gamma - 1)/\beta$. Now define $T_0 = 0$ and

$$T_j = \frac{d}{d - \beta} T_{j-1} + \frac{d - \gamma(j + 2\ell)}{d - \beta}$$

if $1 \leq j \leq 2d - 2\ell$ so that

$$d(1 - T_j + T_{j-1}) + \beta T_j = \gamma(j + 2\ell)$$

for $1 \leq j \leq 2d - 2\ell$. An easy inductive proof shows that $T_j \leq R_j$ for all j. Thus

$$T_{2d-2\ell} \leq R_{2d-2\ell} \leq d(2\gamma - 1)/\beta.$$

Define $t_j = T_j - T_{j-1}$ for $j = 1, 2, \ldots, 2d - 2\ell$. For $1 \leq j \leq 2d - 2\ell$,

$$\begin{aligned} d(1 - t_j) + \beta(t_1 + t_2 + \cdots + t_j) &= d(1 - T_j + T_{j-1}) + \beta T_j \\ &= \gamma(j + 2\ell). \end{aligned}$$

Also

$$T_{2d-2\ell} = t_1 + t_2 + \cdots + t_{2d-2\ell} \le d(2\gamma - 1)/\beta.$$

This means that $t_1, t_2, \ldots, t_{2d-2\ell}, \gamma$ is a feasible solution to LP$_3$. □

LEMMA 7.3. *If $t_1, t_2, \ldots, t_{2d-2\ell}, \gamma$ is a solution to* LP$_3$, *then*

$$\gamma \ge \beta\left[1 + 1/(2e^{2\beta} - 1)\right].$$

Proof. We have $d(1 - t_1) + \beta t_1 = \gamma(1 + 2\ell)$, i.e.,

$$t_1 = \frac{d - \gamma(1 + 2\ell)}{d - \beta}.$$

For $j = 2, 3, \ldots, 2d - 2\ell$,

$$d(1 - t_j) + \beta \cdot \sum_{i=1}^{j} t_i = \gamma \cdot (j + 2\ell),$$

$$d(1 - t_{j-1}) + \beta \cdot \sum_{i=1}^{j-1} t_i = \gamma \cdot (j - 1 + 2\ell).$$

Subtracting, $d(t_{j-1} - t_j) + \beta t_j = \gamma$, or

$$t_j = \frac{d}{d - \beta} t_{j-1} - \frac{\gamma}{d - \beta},$$

$j = 2, 3, \ldots, 2d - 2\ell$. An easy proof verifies that

$$t_j = \frac{\gamma}{\beta} - \left(\frac{\gamma}{\beta} + 2\gamma\frac{\ell}{d} - 1\right)\left(\frac{d}{d - \beta}\right)^j$$

for $j = 1, 2, \ldots, 2d - 2\ell$. We now use this assignment in the last constraint of LP$_3$ to get

$$d + \gamma(2d - 2\ell) + \beta\left(1 - \frac{\gamma}{\beta} - 2\gamma\frac{\ell}{d}\right)\sum_{i=1}^{2d-2\ell}\left(\frac{d}{d - \beta}\right)^i \le \gamma \cdot 2d.$$

But

$$\sum_{i=1}^{2d-2\ell}\left(\frac{d}{d - \beta}\right)^i = \frac{d}{\beta} \cdot \left[\left(\frac{d}{d - \beta}\right)^{2d-2\ell} - 1\right] = \frac{d}{\beta}(D - 1)$$

where $D = (d/(d - \beta))^{2d-2\ell}$. Therefore,

$$d + \gamma(2d - 2\ell) + \beta\left(1 - \frac{\gamma}{\beta} - 2\ell\frac{\gamma}{d}\right)\left[\frac{d}{\beta}(D - 1)\right] \le \gamma(2d),$$

$$d + \gamma(2d - 2\ell) + d\left(1 - \frac{\gamma}{\beta} - 2\ell\frac{\gamma}{d}\right)(D - 1) \le \gamma(2d),$$

$$d(1 + D - 1) + \gamma(-2\ell - \left(\frac{d}{\beta} + 2\ell\right)(D - 1)) \le 0,$$

$$dD \leq \gamma \left(2\ell + (D-1)\left(\frac{d}{\beta} + 2\ell\right)\right),$$

$$\gamma \geq \frac{Dd}{2\ell + (D-1)(\frac{d}{\beta} + 2\ell)} = \frac{Dd\beta}{2\ell\beta + (D-1)(d + 2\ell\beta)}$$

$$= \frac{Dd\beta}{2\ell\beta + Dd - d + D2\ell\beta - 2\ell\beta},$$

$$\gamma \geq \beta \left[\frac{Dd}{Dd - d + D(2\ell\beta)}\right] = \beta \left[\frac{Dd - d + D(2\ell\beta) + [d - D(2\ell\beta)]}{Dd - d + D(2\ell\beta)}\right]$$

$$= \beta \left[1 + \frac{d - D(2\ell\beta)}{Dd - d + D(2\ell\beta)}\right] = \beta \left[1 + \frac{1 - 2\beta D\ell/d}{D(1 + 2\beta\ell/d) - 1}\right].$$

Now

$$D = \left(\frac{d}{d-\beta}\right)^{2d-2\ell} = \left(1 + \frac{\beta}{d-\beta}\right)^{2d-2\ell} \leq e^{\beta(2d-2\ell)/(d-\beta)}.$$

Because $(2d - 2\ell)/(d - \beta) \leq 2$, $D \leq e^{2\beta}$. But

$$\gamma \geq \beta \left[1 + \frac{1 - 2\beta\ell D/d}{D(1 + 2\beta\ell/d) - 1}\right].$$

As D increases, the bracketed quantity decreases. Therefore,

$$\gamma \geq \beta \left[1 + \frac{1 - 2\beta\ell e^{2\beta}/d}{e^{2\beta} - 1 + e^{2\beta}(2\beta\ell/d)}\right].$$

Because $d \geq 4\beta e^{2\beta}\ell$, $2\beta\frac{\ell}{d}e^{2\beta} \leq 1/2$. Thus

$$\gamma \geq \beta \left[1 + \frac{1 - 1/2}{e^{2\beta} - 1 + 1/2}\right] = \beta \left[1 + \frac{1}{2e^{2\beta} - 1}\right]. \qquad \square$$

LEMMA 7.4. *The optimal value of* LP_1 *is at least*

$$\beta\left[1 + 1/(2e^{2\beta} - 1)\right].$$

Proof. The lemma follows from Lemmas 7.1, 7.2, and 7.3. □

Proof of Lemma 2.4. If Lemma 2.4 is not true, then

$$d + c_{i-1} \sum_{s=1}^{Q-1} p_s w_s < c_i(2d)$$

and

$$d(1-p_h)+c_{i-1}\sum_{s=1}^{h} p_s w_s < c_i \sum_{s=1}^{h} w_s$$

for all h, $1 \le h \le Q-1$. In this case, there is a $C < c_i$ such that

$$d + c_{i-1}\sum_{s=1}^{Q-1} p_s w_s \le C(2d)$$

and

$$d(1-p_h)+c_{i-1}\sum_{s=1}^{h} p_s w_s \le C \sum_{s=1}^{h} w_s$$

for all h, $1 \le h \le Q-1$. This contradicts Lemma 7.4 if we set $\beta = c_{i-1}$ and $\gamma = C$, since then

$$\beta\left[1+1/(2e^{2\beta}-1)\right] = c_i. \qquad \square$$

REFERENCES

[BBKTW] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, *On the power of randomization in online algorithms*, Proc. 22nd Annual ACM Symposium on Theory of Computing, 1990, pp. 379–386.

[BCR] R. A. Baeza-Yates, J. C. Culberson, and G. J. E. Rawlins, *Searching with Uncertainty*, Tech. Report, University of Waterloo, 1987.

[BLS] A. Borodin, N. Linial, and M. Saks, *An optimal on-line algorithm for metrical task systems*, Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 373–382.

[BRS] A. Blum, P. Raghavan, and B. Schieber, *Navigating in unfamiliar geometric terrain*, Proc. 23rd Annual ACM Symposium on Theory of Computing, 1991.

[CKPV] M. Chrobak, H. J. Karloff, T. Payne, and S. Vishwanathan, *New results on server problems*, SIAM J. Discrete Math., 4 (1991), pp. 172–181.

[CL] M. Chrobak and L. Larmore, *An optimal on-line algorithm for the server problem on trees*, SIAM J. Comput., 20 (1991), pp. 144–148.

[FFKRRV] A. Fiat, D. P. Foster, H. J. Karloff, Y. Rabani, Y. Ravid, and S. Vishwanathan, *Competitive algorithms for layered graph traversal*, Proc. 32nd Annual IEEE Symposium on Foundations of Comp. Sci., 1991, pp. 288–297.

[FKLMSY] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, *Competitive paging algorithms*, J. Algorithms, 12 (1991), pp. 685–699.

[FRR] A. Fiat, Y. Rabani, and Y. Ravid, *Competitive k-server algorithms*, Proc. 31st Annual IEEE Symposium on Foundations of Comp. Sci., 1990, pp. 454–463; J. Comput. System Sci. (to appear).

[Ge] E. Grove, *The harmonic k-server algorithm is competitive*, Proc. 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 260–266.

[KMMO] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, *Competitive randomized algorithms for non-uniform problems*, Proc. 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 301–309.

[LST] L. Lovász, M. Saks, and W. T. Trotter, *An online graph coloring algorithm with sublinear performance ratio*, Discrete Math. (1989), pp. 319–325.

[MMS] M. S. MANASSE, L. A. MCGEOCH, AND D. D. SLEATOR, *Competitive algorithms for on-line problems*, J. Algorithms, 11 (1990), pp. 208–230.

[MS] L. A. MCGEOCH AND D. D. SLEATOR, *A strongly competitive randomized paging algorithm*, Algorithmica, 6 (1991), pp. 816–825.

[PY] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Shortest paths without a map*, Lecture Notes in Comp. Sci. 372, Springer-Verlag, New York, 1989, pp. 610–620.

[RS] P. RAGHAVAN AND M. SNIR, *Memory versus randomization in on-line algorithms*, Lecture Notes in Comp. Sci. 372, Springer-Verlag, New York, 1989, pp. 687–703.

[ST] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

[Vn] S. VISHWANATHAN, *Randomized online graph coloring*, Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, 1990; J. Algorithms (to appear).

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 313–323, April 1994

007

AN ALPHABET INDEPENDENT APPROACH TO TWO-DIMENSIONAL PATTERN MATCHING[*]

AMIHOOD AMIR[†], GARY BENSON[‡], AND MARTIN FARACH[§]

Abstract. There are many solutions to the string matching problem that are strictly linear in the input size and independent of alphabet size. Furthermore, the model of computation for these algorithms is very weak: they allow only simple arithmetic and comparisons of equality between characters of the input. In contrast, algorithms for two-dimensional matching have needed stronger models of computation, most notably assuming a totally ordered alphabet. The fastest algorithms for two-dimensional matching have therefore had a logarithmic dependence on the alphabet size. In the worst case, this gives an algorithm that runs in $O(n^2 \log m)$ with $O(m^2 \log m)$ preprocessing.

The authors show an algorithm for two-dimensional matching with an $O(n^2)$ text-scanning phase. Furthermore, the text scan requires no special assumptions about the alphabet, i.e., it runs on the same model as the standard linear-time string-matching algorithm. The pattern preprocessing requires an ordered alphabet and runs with the same alphabet dependency as the previously known algorithms.

Key words. multidimensional matching, period, string

AMS subject classifications. 68Q05, 68Q20, 68Q25

1. Introduction. The classical *string-matching problem* has as its input a *text* string T of length n and a *pattern* string P of length m. The elements in the text and pattern are taken from an alphabet set Σ and σ_P is the number of distinct characters in pattern P, so in particular, $\sigma_p \leq \min\{|\Sigma|, m\}$. We will in general drop the subscript P and simply refer to σ. The output is all text locations i where there is a character-by-character match with the pattern, i.e., $T[i + j - 1] = P[j]$, $j = 1, \ldots, m$.

String matching is one of the most widely studied problems in computer science [13]. Fischer and Paterson [12] gave a convolutions-based solution of time complexity $O(n \log m \log \sigma)$ word operations ($O(n \log m \log \log m \log \sigma)$ bit operations). Karp, Miller, and Rosenberg [17] gave a parallelizable label-doubling algorithm with complexity $O(n \log m)$. Knuth, Morris, and Pratt [19] gave the first linear-time solution. A heuristically improved algorithm was presented by Boyer and Moore [10]. Galil and Seiferas [15] showed a real-time algorithm using a constant number of registers. The Knuth, Morris, and Pratt and Galil and Seiferas algorithms have time complexity $O(n)$, are alphabet independent, and use a weak model of computation where only equality of symbols is tested.

Karp and Rabin [18] devised a *randomized* linear-time algorithm in a stronger arithmetic model. They generate a large random prime number as well as use arithmetic operations (e.g., multiplication, modulo) on the characters. Vishkin [24] introduced a deterministic sampling scheme that allowed using the "signature" idea in a deterministic weak model.

In recent years there has been growing interest in multidimensional pattern matching, largely motivated by problems in low-level image processing [23]. Various algorithms exist for the *exact two-dimensional matching* problem. The exact two-dimensional matching problem is defined similarly to the string-matching problem but the text and pattern are rectangular matrices rather than strings. For simplicity's sake we assume that T is an $n \times n$ matrix and P is an $m \times m$ matrix, although our results apply to rectangular matrices as well.

[*] Received by the editors February 18, 1992; accepted for publication (in revised form) November 20, 1992.

[†] College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280; (404) 853-0083 (amir@cc.gatech.edu). Partially supported by National Science Foundation grant IRI-90-13055.

[‡] Department of Computer Science, University of Maryland, College Park, Maryland 20742; (301) 405-2715 (benson@cs.umd.edu). Partially supported by National Science Foundation grant IRI-90-13055.

[§] Center of Discrete Mathematics and Theoretical Computer Science, Box 1179, Rutgers University, Piscataway, New Jersey 08855; (908) 932-5928 (farach@dimacs.rutgers.edu). Supported by Center for Discrete Mathematics and Theoretical Computer Science under National Science Foundation contract STC-88-09648.

Baker [8] and, independently, Bird [9] used the Aho and Corasick [1] dictionary matching algorithm to obtain a $O(n^2 \log \sigma)$ algorithm for the exact two-dimensional matching problem. Their model requires a totally ordered alphabet (since it uses the Aho and Corasick algorithm as a subroutine), and so the time is dependent on the alphabet size. For an unbounded alphabet, their algorithm's time is $O(n^2 \log m)$. Two other algorithms for exact two-dimensional matching appear in [6] and [4]. They both use subword trees and run in time $O(n^2 \log \sigma)$. Note that while these algorithms require no arithmetic operations on the characters, they all assume a total ordering on their alphabets and make order comparisons in addition to checking equality of characters. A convolutions-based method was suggested by Amir and Landau [5]. There the time is $O(\sigma n^2 \log m)$ word operations in an arithmetic model (or also $O(n^2\sqrt{m} \log m)$). The Karp and Rabin algorithm also generalizes to two dimensions but, as we noted before, it is a randomized algorithm with a relatively powerful arithmetic model. Other algorithms with good average case complexities and strong computational models are given in [26] and [7].

In this paper we present what is, to our knowledge, the first deterministic algorithm for two-dimensional exact matching where the text scanning is alphabet independent and thus truly linear. Moreover, our algorithm is comparison based using a weak model of computation. During the text scan, the only character comparisons made are of the equality type, thus the model is weaker than in the above-mentioned two-dimensional matching algorithms. As opposed to previous algorithms, our algorithm is inherently two-dimensional, and uses a novel technique in two-dimensional matching—*two-dimensional periodicity*, as introduced by Amir and Benson in [2].

1.1. The two-dimensional periodicity idea. A periodic pattern contains locations, other than the origin, where the pattern can be superimposed on itself without mismatch. Suppose our pattern is *non-periodic*, i.e., there are no such locations, other than the origin. We could then narrow down the number of *potential candidates* for a pattern appearance in the text in a fashion that ensures that all such candidates are "sufficiently far" from each other. Verification of a candidate could then be done in the naive character-by-character comparison, but the time would still be linear because the candidates do not overlap.

The problem with implementing this idea is that there is no guarantee that the pattern is non-periodic. Indeed it has been shown [2] that there are four different types of two-dimensional periodicity and that a pattern may contain many locations where it can superimpose on itself without mismatch. Moreover, it is not possible to subdivide all patterns into non-periodic subunits, as is the case with one-dimensional strings. In this paper, we make use of the very strong property that superimposable patterns can not disagree in the area of overlap, and we present a new method for exploiting the pattern's periodicity.

In contrast, previous algorithms have, to a lesser or greater degree, shared a common weakness. They all treat a matrix as a *set* of rows, rather than as a sequence of rows. That is, they only consider periodicity one dimension at a time. Thus, while exploiting periodicity *within* rows, information about periodicity *among* rows is disregarded. The extra log factor can be seen as a way to recompute information that was discarded in earlier stages of the algorithm. Our unified approach to two-dimensional periodicity allows us to use all periodicity information throughout the text-scanning algorithm.

Our algorithm consists of a *pattern analysis* stage and a *text-scanning* stage. In the pattern analysis we construct a WITNESS array that allows a constant time decision of whether two overlapping pattern appearances conflict. This stage is done in time $O(m^2 \log \sigma)$, $O(m^2 \log m)$ in the worst case, and assumes an ordered alphabet. Note that very recently, there have been several advances in two-dimensional string matching. In [14] and independently in [3], it was shown how to compute a witness table in $O(m^2)$ using the unordered alphabet model of computation.

The text-scanning stage has two phases, the *compatibility phase* and the *verification phase*. We begin by assuming that the pattern could occur anywhere in the text. In the compatibility phase we eliminate candidate locations until all remaining candidates agree on the expected text characters. We are left with potential candidates that are all *compatible* with each other. In the verification phase we verify which of these potential candidates are indeed a match. The entire text-scanning stage is done in time $O(n^2)$.

The paper is organized as follows. The pattern analysis is described in §2. Section 3 consists of the text scan.

2. Pattern preprocessing. The idea of array overlap or *periodicity* and the pattern preprocessing algorithm are given in [2]. For completeness, we review the algorithm here. Our goal is to determine where two copies of an array A can overlap without conflict. Such sites are called *sources* (Fig. 1). For each location that is not a source, there exists a *witness* that proves that the overlapping copies of A mismatch.

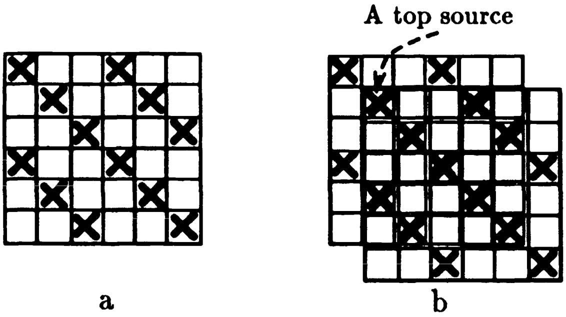


FIG. 1. (a) *An array A.* (b) *A overlaps itself without a mismatch.*

Given two copies of an $m \times m$ array $A[1, \ldots, m; 1, \ldots, m]$ one directly on top of the other, the two copies are said to be *in register* when all of the corresponding elements in the area of overlap contain the same symbol. Clearly, A is in register with itself when $A[1, 1]$ is aligned with $A[1, 1]$. If we can slide the upper copy over the lower copy to a point where the copies are again in register, then at least one of the corner elements $A[1, 1]$ or $A[m, 1]$ in one copy overlaps an element of the other copy. If the overlapping corner is $A[1, 1]$ then we have a *top source*. Otherwise, we have a *bottom source*.

We want to fill out two WITNESS arrays. For each location $A[i, j]$, TOP-WITNESS$[i, j]$ $= (m + 1, m + 1)$ if A is in register with itself when element $A[1, 1]$ overlaps element $A[i, j]$. Otherwise, TOP-WITNESS$[i, j] = (r, c)$ where (r, c) identifies some mismatch. Specifically $A[r, c] \neq A[i + r - 1, j + c - 1]$ (Fig. 2). BOTTOM-WITNESS$[i, j]$ is filled out similarly except element $A[m, 1]$ overlaps element $A[i, j]$.

2.1. The pattern preprocessing algorithm. Our pattern preprocessing algorithm (Algorithm A) makes use of two algorithms (Algorithms 1 and 2) from [21] which are themselves variations of the KMP algorithm [19] for string matching. Algorithm 1 takes as input a pattern string w of length m and builds a table $lppattern[1, \ldots, m]$ where $lppattern[i]$ is the length of the longest prefix of w starting at w_i. Algorithm 2 takes as input a text string t of length n and the table produced by Algorithm 1 and produces a table $lptext[1..n]$ where $lptext[i]$ is the length of the longest prefix of w starting at t_i.

The idea behind Algorithm A is the following. We convert the two-dimensional problem into a problem on strings (Fig. 3). Let the array A be processed column by column and suppose we are processing column j. Assume we can convert the block $A[1..m, j..m]$ into a string

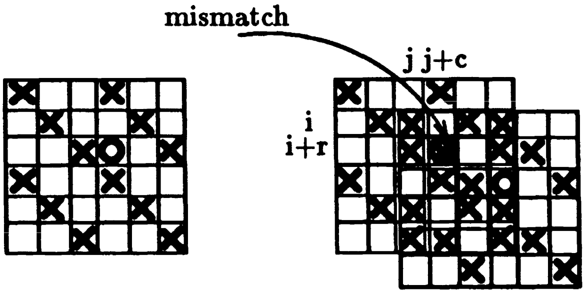


FIG. 2. *The WITNESS table gives the location of a mismatch (if one exists) for two overlapping copies of the pattern. Here TOP-WITNESS*$[i, j] = (r + 1, c + 1)$.

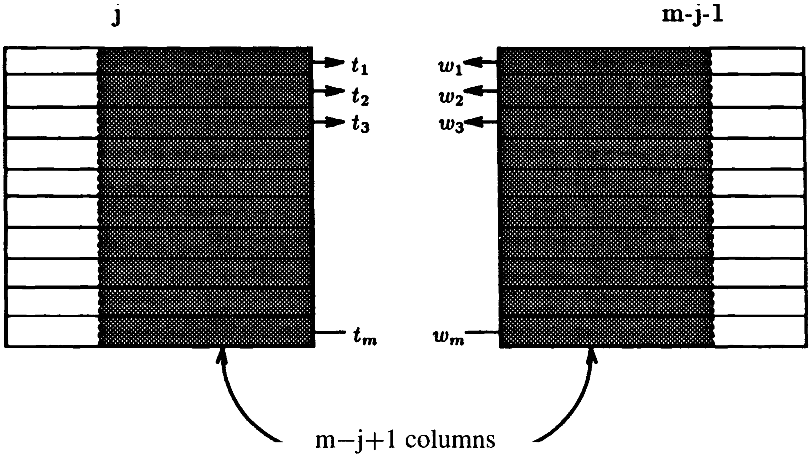


FIG. 3. *Representing a block of the array by a string. For the preprocessing algorithm,* $T_j = t_1, \ldots, t_m$ *is the text and* $W_j = w_1, \ldots, w_m$ *is the pattern.*

$T_j = t_1, \ldots, t_m$ where t_i represents the suffix of row i starting in column j. This will serve as the text string. Assume also that we can convert the block $A[1..m, 1..m - j + 1]$ into a string $W_j = w_1, \ldots, w_m$ where w_i represents the prefix of row i of length $m - j + 1$. This will serve as the pattern string. Now, use Algorithm 1 to produce the table *lppattern* for W_j and Algorithm 2 to produce the table *lptext* for T_j. If the longest prefix of the pattern in the text starting at t_i runs through the last row of the text ($lptext[i] = m - i + 1$), then $A[i, j]$ is a source. If the longest prefix stops before the last row ($lptext[i] < m - i + 1$), then there is a mismatch between the prefix of row $lptext[i]$ and the suffix of row $i + lptext[i]$. We need merely locate the mismatch to obtain the witness. In order to treat the suffix and prefix of a row as a single character, we will build a *suffix tree* for the array.

A suffix tree is a compacted trie of the suffixes of a string ([22], [25]). The suffix tree is perhaps the most widely used data structure in string matching. A thorough description of suffix trees and their properties appears in [11]. We note that since a suffix tree is a trie, each node v has associated with it some string $S(v)$. In [20], it was pointed out that if l is the least common ancestor (LCA) of two nodes v and w, then $S(l)$ is the longest common prefix of $S(v)$ and $S(w)$. In [16], an algorithm was given that preprocesses a tree in linear time and answers LCA queries in constant time. Thus a suffix tree, in conjunction with LCA queries, is a powerful tool for comparing the substrings of a string.

ALGORITHM A. *For building witness array*

Step A.1: Build a suffix tree by concatenating the rows of the array. Preprocess the suffix tree for least common ancestor queries in order to answer questions about the length of the common prefix of any two suffixes.

Step A.2: For each column j, fill out TOP-WITNESS for column j:

- Step A.2.1: Use Algorithm 1 to construct the table *lppattern* for $W_j = w_1 \dots w_m$. Character w_i is the *prefix* of row i of length $m - j + 1$. We can answer questions about the equality of two characters by consulting the suffix tree. If the length of the common prefix of the two characters is at least $m - j + 1$ then the characters are equal.
- Step A.2.2: Use Algorithm 2 to construct the table *lptext* for $T_j = t_1 \dots t_m$. Character t_i is the *suffix* of row i starting in column j (also of length $m - j + 1$). Again we test for equality by reference to the suffix tree.
- Step A.2.3: For each row i, if $lptext[i] = m - i + 1$ then we have found a source and TOP-WITNESS$[i, j] = (m + 1, m + 1)$ otherwise, using the suffix tree, compare the suffix of row $i + lptext[i]$ starting in column j with the prefix of row $lptext[i]$. The length l of the common prefix will be less than $m - j + 1$, and TOP-WITNESS$[i, j] = (lptext[i], l + 1)$.

Step A.3: Repeat step 2 for BOTTOM-WITNESS by building the automatons and processing the columns from the bottom up.

THEOREM 2.1. *Algorithm* A *runs in time* $O(m^2 \log \sigma)$.

Proof. The suffix tree construction [25] takes time $O(m^2 \log \sigma)$ while the preprocessing for least common ancestor queries [16] can be done in time linear in the size of the array. Queries to the suffix tree are processed in constant time. The tables *lppattern* and *lptext* can be constructed in time $O(m)$ [21]. For each of m columns, we construct two tables so the total time for steps 2 and 3 is $O(m^2)$. The total complexity of the pattern preprocessing is therefore $O(m^2 \log \sigma)$. □

3. Text processing. Text processing is accomplished in two stages: candidate consistency and candidate verification. A *candidate* is a location in the text where the pattern may occur. We denote a candidate with origin at text location $T[r, c]$ by (r, c). We say that two candidates (r, c) and (x, y) are *consistent* if they expect the same text characters in their region of overlap (two candidates with no overlap are trivially consistent). In terms of witnesses, two candidates are consistent if they have no witness, i.e., if $r \leq x$ and $c \leq y$ then *TOP-WITNESS*$[x - r + 1, y - c + 1] = (m + 1, m + 1)$. If $r > x$ and $c \leq y$ then *BOTTOM-WITNESS*$[m - r + x, y - c + 1] = (m + 1, m + 1)$. We use the shorthand $(r, c) \sim (x, y)$ to mean that the candidates (r, c) and (x, y) are consistent. If the two candidates are inconsistent, then we write $(r, c) \not\sim (x, y)$.

Initially, we have no information about the text and therefore all text locations are candidates. However, not all text locations are consistent. During the candidate consistency phase, we eliminate candidates until all remaining candidates are pairwise consistent. During the candidate verification phase, we check the candidates against the text to see which candidates represent actual occurrences of patterns. We exploit the consistency of the surviving candidates to rule out large sets of candidates with single text comparisons (since all consistent candidates expect the same text character).

3.1. Candidate consistency. As stated above, the goal of the *candidate consistency algorithm* presented in this subsection is to produce a set of candidates for the given text such that the candidates are all consistent.

We begin with some transitivity lemmas for the $\sim$ relation.

LEMMA 3.1. *For any* $1 \le r_1 \le r_2 \le r_3 \le n$ *and for any* $1 \le c_1 \le c_2 \le c_3 \le n$, *if* $(r_1, c_1) \sim (r_2, c_2)$ *and* $(r_2, c_2) \sim (r_3, c_3)$, *then* $(r_1, c_1) \sim (r_3, c_3)$.

Proof. Suppose that $(r_1, c_1) \not\sim (r_3, c_3)$. Then, there exists an $x \le m - r_3 + r_1$ and a $y \le m - c_3 + c_1$ such that $P[x, y] \ne P[x + r_3 - r_1, y + c_3 - c_1]$. But $r_3 \ge r_2$ so $x + r_3 \ge r_2$ and $m \ge x + r_3 - r_1 \ge r_2 - r_1$. Similarly, $m \ge y + c_3 - c_1 \ge c_2 - c_1$. Since $(r_1, c_1) \sim (r_2, c_2)$, we have that $P[x + r_3 - r_1, y + c_3 - c_1] = P[x + r_3 - r_2, y + c_3 - c_2]$. A similar argument shows that $P[x, y] = P[x + r_3 - r_2, y + c_3 - c_2]$ since $(r_3, c_3) \sim (r_2, c_2)$. We conclude that $P[x, y] = P[x + r_3 - r_1, y + c_3 - c_1]$. This is a contradiction. Therefore $(r_3, c_3) \sim (r_1, c_1)$. □

LEMMA 3.2. *For any* $1 \le r_1 \le r_2 \le r_3 \le n$ *and for any* $1 \le c_3 \le c_2 \le c_1 \le n$, *if* $(r_1, c_1) \sim (r_2, c_2)$ *and* $(r_2, c_2) \sim (r_3, c_3)$, *then* $(r_1, c_1) \sim (r_3, c_3)$.

Proof. The proof is analogous to that of Lemma 3.1. □

3.1.1. A one-dimensional consistency algorithm.

Let c be some column of the text. Initially, all positions in this column are candidates. We would like to remove candidates until all candidates within the column are consistent. Further, we would like to preserve any candidate that might actually represent an occurrence of the pattern in the text. Thus, we will only remove candidates when we find some specific text location with which they mismatch. The idea of Algorithm B is the following. Suppose we have eliminated inconsistent candidates from the last i rows of column c. The surviving candidates are placed on a list. Notice that by Lemma 3.1, if the candidate in row $n - i$ is consistent with the top candidate on the list, it is consistent with all of them. This check takes constant time using the witness array. This principle is used to produce an $O(n)$ algorithm for column consistency.

ALGORITHM B. *Eliminate inconsistent candidates within a column*

```
Step B.1: Get column number, $c$.
Step B.2: We create a doubly linked list, $S$, of consistent candidates in column
$c$. Initialize $S$ by adding candidate $(n - m + 1, c)$ to the top of $S$.
Step B.3: For row $r = n - m$ to 1 do:
    Step B.3.1: Let $(x, c)$ be the top candidate in $S$. Test if candidates $(r, c)$
    and $(x, c)$ are consistent by reference to the witness arrays:
      * If $(r, c) \sim (x, c)$, then add $(r, c)$ to the top of $S$.
```

If the two candidates under consideration are consistent, then they need not be compared with any other candidates on S. This is because, by Lemma 3.1, consistency within a single column is transitive.

```
      * If $(r, c) \not\sim (x, c)$ then use the witness character in the text to eliminate
        at least one of the candidates. If $(x, c)$ is eliminated, remove it from
        $S$ and if $(r, c)$ is not eliminated, repeat step B.3.1 with the new top
        candidate in $S$. If no candidates remain in $S$, add $(r, c)$ to $S$.
```

Clearly, if the two candidates are inconsistent, they can't both match the text. Thus the inappropriate one is eliminated.

```
Step B.4.3: Return $S$.
```

THEOREM 3.3. *Algorithm* B *is correct and runs in time* $O(n)$.

Proof. The correctness of the algorithm follows largely from the comments within the algorithm and from Lemma 3.1.

For the complexity bound, note that S can be initialized in constant time. For each row r in the for loop, there is at most one successful test of consistency. For each unsuccessful test, a candidate is eliminated, either the candidate (r, c) or the top candidate in S. Since the number of candidates is bounded by n the total time is $O(n)$. □

3.1.2. A two-dimensional consistency algorithm. We use the above algorithm as an initial "weeding out" of candidates so that we get a list for each column of consistent candidates. In the two-dimensional consistency algorithm, we start with the rightmost column, which we know to be consistent, and add one column at a time from right to left. We will maintain the following loop invariant: $P(i) \equiv$ the candidates remaining in columns $i, \ldots, n$ are all pairwise consistent.

No candidates can occur in columns $n - m + 2, \ldots, n$ so $P(n - m + 2), \ldots, P(n)$ are trivially satisfied. As noted above, by calling Algorithm B with value $n - m + 1$ we are assured of $P(n - m + 1)$. The approach of the algorithm below is to quickly ensure $P(i)$ once $P(i + 1)$ is known. When $P(1)$ holds, we are done. We use a similar idea to that of Algorithm B. We first have a phase where we make sure that each candidate is consistent with all candidates above and to the right. A symmetric phase makes sure that candidates below and to the right are consistent, thus assuring $P(i)$. To reduce the work, we note that during the first phase, we need only compare a candidate on column i with the leftmost surviving candidate in each row above it. To further reduce the work, once a candidate in column i is found to be consistent with candidates above it, all lower candidates in column i are also consistent (see Fig. 4).

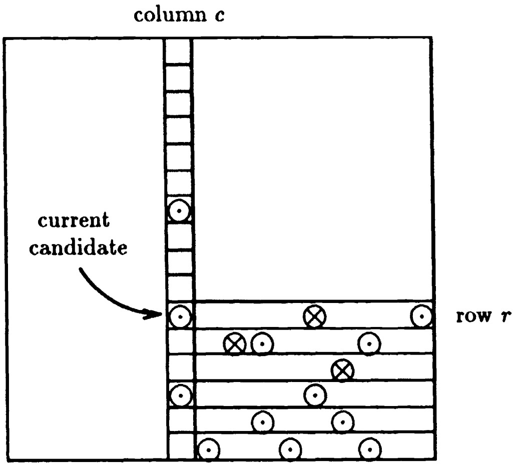


FIG. 4. *In the bottom-up scan, the current candidate in column c need only be tested against the leftmost candidates (marked by ⊗) in rows $r + m, \ldots, r$ that have not already been tested by candidates below c.*

```
ALGORITHM C. Candidate consistency
    Step C.1: For $i \leftarrow 1$ to $n - m + 1$ do $C_i \leftarrow$ Call Algo B($i$)
    Step C.2: For $i \leftarrow 1$ to $n - m + 1$ do initialize $R_i$ to be an empty list of candidates
    for each row $i$.
    Step C.3: Put the candidates on $C_{n-m+1}$ onto their appropriate $R_i$ lists.
    Step C.4: For $i \leftarrow n - m$ downto 1 do
  Add one row at a time, making sure that it is consistent with all candidates added so far.
        Step C.4.1: Call Bottom-Up($i$)
  Make sure that all candidates in column $i$ are consistent with all candidates below them
  in columns $i + 1, \ldots, n$.
        Step C.4.2: Call Top-Down($i$)
  Make sure that all candidates in column $i$ are consistent with all candidates above them
  in columns $i + 1, \ldots, n$.
```

```
        Step C.4.3: Add surviving candidates from column $i$ to the appropriate
        $R_j$ lists.
```

We describe procedure Bottom-Up only, since procedure Top-Down is symmetric.

```
Procedure C1. Bottom-Up(c)
    Step C1.1: Initialize: $cur$ gets bottom value from $C_c$. $row \leftarrow n - m + 1$ is a
    pointer to the last row compared so far.
    Step C1.2: While not at the top of $C_c$ do
        Step C1.2.1: If $cur$ is consistent with leftmost item on $R_{row}$ or $R_{row}$ is
        empty, then $row \leftarrow row - 1$.
```

We compare the current candidate with the leftmost candidate in some row row below it. If they are consistent, then by Lemma 3.1, all candidates above cur on C_c are also consistent with all candidates on R_{row}, *even if cur is later deleted as inconsistent with another candidate.* We need not consider that row again.

```
        Step C1.2.2: If $cur$ is not consistent with leftmost item on $R_{row}$, then find
        a witness to their inconsistency. Check which of them has a mismatch
        against the text. If the leftmost item on $R_{row}$ has a mismatch, remove
        that candidate from its list. If $cur$ has a mismatch, set $cur$ to the next
        item above $cur$ on $C_c$.
```

We remove the candidate that has a mismatch against the text. If the item in R_{row} is removed, then we still need to check if cur is consistent with the remaining candidates in that row. Thus, we don't need to update any pointers. Otherwise, if cur is removed, we move up in C_c. We don't need to change row because of the comment above. None of the rows below row need to be compared against the new candidate cur since we already know they are consistent.

```
        Step C1.2.3: If the $row$ counter points to a row above $cur$'s row, set $cur$
        to the next candidate above $cur$ in $C_c$.
```

THEOREM 3.4. *Algorithm* C *is correct and runs in* $O(n^2)$.

Proof. As in Algorithm B, no candidate is removed unless a mismatch is found against the text. Therefore, no valid candidates are removed.

To show that at the end of the algorithm, only mutually consistent candidates are left on the R_i lists (and on the C_i), we pick two arbitrary surviving candidates (r_1, c_1) and (r_2, c_2) such that $c_1 < c_2$. We have two cases.

Case $r_1 \leq r_2$: We show this case by induction. Suppose that after processing column $c_1 + 1$ that $P(c_1 + 1)$ holds. The base case is true by Theorem 3.3. Let (r_2, c') be the leftmost candidate such that $c' > c_1$ and c' appears on R_{r_2} after processing column c_1. By Lemma 3.1, we need only show that $(r_1, c_1) \sim (r_2, c')$ since $(r_2, c') \sim (r_2, c_2)$.

Let (r', c_1) be the last candidate with which (r_2, c') was compared during *Bottom-Up*(c_1).

CLAIM 3.4.1. $r' \geq r_1$ *and* $(r', c_1) \sim (r_2, c')$.

Proof. Suppose that $(r', c_1) \not\sim (r_2, c')$. Then we either delete (r', c_1) or (r_2, c') from the candidate list. If we remove (r_2, c') from the list, then we would compare the next candidate on R_{r_2} with (r', c_1), thus violating the assumption that (r_2, c') was the leftmost candidate compared with a c_1 candidate. If we remove (r', c_1), then we would compare (r_2, c') with the next candidate above (r', c_1), thus violating the assumption that (r', c_1) was the last candidate on column c_1 with which (r_2, c') was compared.

To show that $r' \geq r_1$ we observe that if $r_1 > r'$, then we couldn't have compared (r_2, c') with (r', c_1) without first comparing (r_1, c_1) with (r_2, c'). Since they both survived, they would have had to have been consistent. But then we never would have compared (r_2, c') with (r', c_1) at all. □

Finally, we know that $(r_1, c_1) \sim (r', c_1)$, $(r', c_1) \sim (r_2, c')$, $(r_2, c') \sim (r_2, c_2)$ and that $r_1 \leq r' \leq r_2$ and that $c_1 \leq c' \leq c_2$. So by Lemma 3.1, we have proved the case.

Case $r_1 > r_2$: This case is very similar to the one above, however, we refer the reader to procedure Top-Down rather than Bottom-Up and Lemma 3.2 rather than Lemma 3.1.

The argument that shows the running time to be $O(n^2)$ is similar to the complexity analysis in Theorem 3.3. We observe that during Bottom-Up (and Top-Down) in each, comparison of candidates results in the removal of a candidate (which can only happen n^2 times in all calls to these procedures), or in the *cur* pointer being decremented (respectively, incremented). This can only happen $O(n)$ time each time Bottom-Up (respectively, Top-Down) is called, and they are each called $O(n)$ times. Therefore the complexity is $O(n^2)$. □

3.2. Candidate verification. All remaining candidates are now mutually consistent. Each text element $t = T[r, c]$ may be contained by several candidates, the *relevant* candidates. However, consistent candidates that share the same text element must agree on the expected character in that element. This leads to the following crucial observation. Every element in T can be labeled as either *true* or *false*, where *true* means that it equals the unique pattern symbol expected by all relevant candidates, and *false* in all other cases. Thus, every text element needs to be compared to a *single* pattern element, and every candidate source that contains a *false* element within it is not a pattern appearance and can be discarded.

The candidate verification algorithm follows:

ALGORITHM D. *Candidate verification*

Step D.1: Mark every text location $T[r, c]$ with a *pattern coordinate pair* $\langle i, j\rangle$, where $\langle i, j\rangle$ are the coordinates of the pattern element $P[i, j]$ with which $T[r, c]$ should be compared.

There may be several options for some locations, namely, the position of the scanned text element relative to each of its relevant candidates. However, any will do since all candidate sources are now consistent. If a location is not contained in any candidate source it is left unmarked. We will later see how this step is implemented (Procedure D1).

Step D.2: Compare each text location $T[r, c]$ with $P[i, j]$, where $\langle i, j\rangle$ is the pattern coordinate pair of $T[r, c]$. If $T[r, c] = P[i, j]$ then label $T[r, c]$ as *true*, else label it *false*.

Step D.3: Flag with a *discard* every candidate that contains a *false* location within its bounds.

This flagging is done by the same method as in step D.1.

Step D.4: Discard every candidate source flagged with a *discard*. The remaining candidates represent all pattern appearances.

Our only remaining task is to show how to mark the text elements with the appropriate pattern coordinate pairs. We adopt the popular sports fan's technique—*the wave.*

Starting at the top (left) of each column (row), a wave is propagated going down (to the right) as follows. The first element stands and waves its pattern coordinate pair, if such exists. This nudges the neighbor below (to the right of) it to jump and raise its own pair. If it does not have a pair, it borrows its antecedent's pair, incrementing by 1 its row (column) coordinate, to adjust for its position relative to the same source. If the pair assigned to some position exceeds the size of the pattern, that position is left unmarked.

Thus in two sweeps of the text, column waves and row waves, each text element is given an appropriate pattern coordinate pair. Details of the wave follow:

Procedure D1. *The wave*

Step D1.1: **Initialization:** Mark every candidate origin with $\langle 1, 1\rangle$.

Step D1.2: **Column Waves:** For each column c, and for all positions r from 1 to n in column c do the following step: If $T[r, c]$ does not have a pair, and $T[r - 1, c]$ has pair $\langle i, j\rangle$ with $i < m$ then assign to $T[r, c]$ the pair $\langle i + 1, j\rangle$.

Step D1.3: **Row Waves:** For each row r, and for all positions c from 1 to n in row r do the following step: If $T[r, c]$ does not have a pair, and $T[r, c-1]$ has pair $\langle i, j\rangle$ with $j < m$ then assign to $T[r, c]$ the pair $\langle i, j+1\rangle$.

A similar version of the wave can be used to flag candidates with *discard.* What is propagated there is the *discard* flag, along with a counter pair to make sure the *discard* flag doesn't get propagated too far. The propagation is bottom-up in the columns and then from right to left within the rows.

THEOREM 3.5. *Algorithm* D *is correct and runs in time* $O(n^2)$.

Correctness. The only non-trivial fact is that the wave correctly marks all elements. We need the following terminology. If (r, c) is a candidate, we refer to the candidate origin $T[r, c]$ as a *source.* Let (r, c) be a candidate containing position $T[r+i, c+j]$. Then j is the *column distance* and i is the *row distance* between $T[r+i, c+j]$ and the source for (r, c). The *column-close* candidates containing location $T[r, c]$ have sources whose column distance to $T[r, c]$ is minimal. The *closest* candidate containing location $T[r, c]$ is the column-close candidate whose source has smallest row distance to $T[r, c]$.

CLAIM 3.5.1. *The pattern coordinate pair marked by Procedure* D1 *in location* $T[r, c]$ *is the pair* $\langle i, j\rangle$ *where* $(r-i+1, c-j+1)$ *is the closest source to* $T[r, c]$.

Proof. By induction on the column distance of the closest source. For column distance 0 the column wave assures that the marked pair is $\langle i, 1\rangle$ where i is the row distance to the closest source $+1$. Assuming that for every text element whose column distance to its closest source is d, the marked pair is correct, it is easy to see that the row wave will ensure correct marking of all elements with column distance $d+1$ to the closest source.

Time. Each of the steps of Algorithm D is easily implementable in time $O(n^2)$. Note that, in each of steps D.1 and D.4, there is a single call to Procedure D1, which clearly takes $O(n^2)$ time. □

4. Conclusion. While string matching is extremely well studied and understood, multidimensional matching has been somewhat neglected. This neglect does not stem from lack of practical motivation but may be attributed to the fact that string-matching techniques do not easily generalize to higher dimensions.

We feel that an inherently multidimensional approach is likely to produce better results. This paper is a step along the way. All previously known algorithms for exact two-dimensional matching pushed string-matching techniques as tools for solving the two-dimensional case. However, none succeeded in achieving results similar to the string-matching case. Our new idea of analyzing periodicity in two dimensions has been useful in improving results of the most basic two-dimensional task—that of exact matching.

REFERENCES

[1] A. AHO AND M. CORASICK, *Efficient string matching*, C. ACM, 18 (1975), pp. 333–340.

[2] A. AMIR AND G. BENSON, *Two-dimensional periodicity and its application*, Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL, 1992, pp. 440–452.

[3] A. AMIR, G. BENSON, AND M. FARACH, *The truth, the whole truth and nothing but the truth: Alphabet independent 2-d witness table construction*, Tech. Rep. GIT-CC-92-51, Georgia Tech, 1992.

[4] A. AMIR AND M. FARACH, *Two dimensional dictionary matching*. Inform. Process. Lett., 44 (1992), pp. 233–239.

[5] A. AMIR AND G. LANDAU, *Fast parallel and serial multidimensional approximate array matching*, Theoret. Comput. Sci., 81 (1991), pp. 97–115.

[6] A. AMIR, G. LANDAU, AND U. VISHKIN, *Efficient pattern matching with scaling*, Proc. of First Symposium on Discrete Algorithms, San Francisco, CA, 1990, pp. 344–357.

[7] R. BAEZA-YATES AND M. RÉGNIER, *Fast algorithms for two dimensional and multiple pattern matching*, Proc. of 2nd Annual Scandinavian Workshop in Algorithmic Theory, SWAT '90, 1990.

[8] T. Baker, *A technique for extending rapid exact-match string matching to arrays of more than one dimension*, SIAM J. Comput., 7 (1978), pp. 533–541.

[9] R. Bird, *Two dimensional pattern matching*, Inform. Process. Lett., 6 (1977), pp. 168–170.

[10] R. Boyer and J. Moore, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[11] M. T. Chen and J. Seiferas, *Efficient and elegant subword tree construction*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., NATO ASI Series F: Computer and System Sciences, 1985, Ch. 12, pp. 97–107.

[12] M. Fischer and M. Paterson, *String matching and other products*, in Complexity of Computation, R. M. Karp, ed., SIAM-AMS Proceedings, 7 (1974), pp. 113–125.

[13] Z. Galil, *Open problems in stringology*, in Combinatorial Algorithms on Words, A. Apostolico, ed., Vol. 12, NATO ASI Series F, 1985, pp. 1–8.

[14] Z. Galil and K. Park, *Truly alphabet-independent two-dimensional pattern matching*, Proc. 33rd IEEE FOCS, 1992.

[15] Z. Galil and J. Seiferass, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[16] D. Harel and R. Tarjan, *Fast algorithms for finding nearest common ancestor*, J. Comput. System Sci., 13 (1984), pp. 338–355.

[17] R. Karp, R. Miller, and A. Rosenberg, *Rapid identification of repeated patterns in strings, arrays and trees*, Symposium on the Theory of Computing, 4 (1972), pp. 125–136.

[18] R. Karp and M. Rabin, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., (1987), pp. 249–260.

[19] D. Knuth, J. Morris, and V. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[20] G. Landau and U. Vishkin, *Efficient string matching in the presence of errors*, Proc. 26th IEEE FOCS, 1985, pp. 126–126.

[21] M. Main and R. Lorentz, *An $O(n \log n)$ algorithm for finding all repetitions in a string*, J. Algorithms, (1984), pp. 422–432.

[22] E. M. McCreight, *A space-economical suffix tree construction algorithm*, J. ACM, 23 (1976), pp. 262–272.

[23] A. Rosenfeld and A. Kak, *Digital Picture Processing*, Academic Press, New York, 1982.

[24] U. Vishkin, *Deterministic sampling - a new technique for fast pattern matching*, SIAM J. Comput., 20 (1991), pp. 303–314.

[25] P. Weiner, *Linear pattern matching algorithm*, Proc. 14 IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.

[26] R. F. Zhu and T. Takaoka, *A technique for two-dimensional pattern matching*, Comm. ACM, 32 (1989), pp. 1110–1120.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 324–334, April 1994

TRADING SPACE FOR TIME IN UNDIRECTED *s*-*t* CONNECTIVITY*

ANDREI Z. BRODER†, ANNA R. KARLIN‡, PRABHAKAR RAGHAVAN§, AND ELI UPFAL¶

Abstract. Aleliunas et al. [20th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1979, pp. 218–223] posed the following question: "The reachability problem for undirected graphs can be solved in log space and $O(mn)$ time [m is the number of edges and n is the number of vertices] by a probabilistic algorithm that simulates a random walk, or in linear time and space by a conventional deterministic graph traversal algorithm. Is there a spectrum of time-space trade-offs between these extremes?" This question is answered in the affirmative for sparse graphs by presentation of an algorithm that is faster than the random walk by a factor essentially proportional to the size of its workspace. For denser graphs, this algorithm is faster than the random walk but the speed-up factor is smaller.

Key words. space-time tradeoff, connectivity testing, parallel random walks

AMS subject classifications. 05C40, 05C85, 60J15, 68Q25

1. Motivation and results.

We consider the problem of s-t connectivity on an undirected graph (USTCON). Given a graph G with n vertices and m edges, and given two vertices s and t of G, we are to decide if s and t are in the same connected component. We are interested in space-bounded algorithms for USTCON, which is an important problem in the study of space-bounded complexity classes [3], [8]. Throughout this paper, we assume that our workspace takes the form of p registers, each capable of storing a $\log n$-bit number.

There are two well-known approaches to solving USTCON: via a deterministic graph search on G (e.g., depth-first search), and via a simulation of a random walk on G [1]. (The standard random walk on G is the stochastic process associated with a particle moving from vertex to vertex according to the following rule: if d_i is the degree of vertex i then the probability of a transition from vertex i to vertex j is $1/d_i$ if $\{i, j\}$ is an edge in G and 0 otherwise.)

The first approach can be implemented to run in time $O(m)$ using space $O(n)$. The latter requires space $O(1)$, and has been shown to decide USTCON in time $O(mn)$ with one-sided error (i.e., if s and t are in the same connected component, the algorithm outputs YES with probability at least 0.5; if they are in different components, the algorithm outputs NO). For both these algorithms, the product of time and space is $O(mn)$.

Given space that is insufficient for depth-first search, can we decide USTCON faster than via a random walk? More precisely, given space $p \leq n$, can we bridge the gap between the depth-first search and the random walk by devising an algorithm that runs in time $O(mn/p)$? Considering the time-space product achieved at the two extremes, this seems a likely conjecture.

In this paper we present an algorithm that runs in time $O(m^2 \log^5 n/p)$ in space $O(p)$. Therefore, for linear-sized graphs (i.e., $m = O(n)$), it achieves the bound conjectured above within a poly-log factor. For denser graphs, our algorithm does not achieve the bound, but it is faster than the random walk for m/p sufficiently small.

The informal description of the algorithm is as follows.

*Received by the editors March 29, 1990; accepted for publication (in revised form) November 30, 1992.

†DEC Systems Research Center, Palo Alto, California 94301.

‡DEC Systems Research Center, Palo Alto, California 94301. Part of this research was done while the author was a research associate at Princeton University. Research supported in part by National Science Foundation grant DCR-8605961 and Office of Naval Research contract N00014-87-K-0467.

§IBM T.J. Watson Research Center, Yorktown Heights, New York 10598.

¶IBM Almaden Research Center, San Jose, California 95120 and Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel. Work at the Weizmann Institute supported in part by a Bat-Sheva de Rothschild Award and by a Revson Career Development Award.

```
Algorithm STConn.
    1. Repeat $O(\log n)$ times
        (a) Choose $p$ random vertices according to the stationary distribution of the random
            walk on $G$. (The stationary distribution of the random walk is $\pi_v = d_v/(2m)$
            where $d_v$ is the degree of vertex $v$.) Call these vertices, together with $s$ and $t$,
            leaders.
        (b) Repeat $O(\log n)$ times: Starting from each leader, take a random walk of length
            $\tau_1 = O(m^2/p^2 \log^3 n)$. If such walks connect two leaders, then mark them and
            all the other leaders known to be connected to them, as belonging to the same
            component. If at any point $s$ and $t$ are marked as being in the same component,
            then stop and report "connected."
    2. Report "probably not connected."
End STConn.
```

A more precise description of the algorithm is given in §3. Clearly the space required is $O(p)$ and the time required is $O(m^2 \log^5 n/p)$. (The leaders are marked via a standard union-find algorithm.) Notice that this algorithm resembles standard search when $p = n$ and the random walk when $p = 0$. (However, throughout this paper we shall assume $p > 0$.)

There are three facts that must be proven in order to show that this algorithm works. The first is to show that a set of p random walks of length τ_1, one from each of the randomly chosen leaders, visits all the vertices of a connected graph with high probability. Otherwise an adversary could choose s and t among those vertices unlikely to be visited from the other leaders and conceivably foil the algorithm. In other words, we need to derive a bound on the expected time required by p parallel and independent random walks to cover the graph, a problem of interest in its own right. Typically, results about graph coverage rely heavily on the long-run behavior of the corresponding Markov chain and its convergence to a limit distribution. Here we must prove something about short-term behavior of the Markov chain and coverage of local neighborhoods in a graph.

The second fact to prove is that if s and t are in the same component and enough leaders are chosen within that component, then with high probability s and t are linked up after a small number of walks from each leader. Coverage of the graph as described above does not ensure linkage, since s and t may be visited only by walks from two disjoint sets of leaders that are never linked. Furthermore, all the vertices in G could be visited by the walks even with s and t in different components.

The third fact to show is that, with high probability, within $O(\log n)$ choices of the set of leaders, the component containing s and t gets enough leaders at least once.

To aid the intuition of the reader, let us consider the case when G is a simple path on n vertices. For p leaders chosen at random, with high probability, the maximum gap between two leaders is no more than $n \ln n/p$; the expected time to cover this maximum gap is $\Theta(n^2 \log^2 n/p^2)$. Hence $O(\log n)$ trials (random walks of length $O(n^2 \log^2 n/p^2)$ from each leader) will almost surely cover all the gaps between them for a total of $\Theta(n^2 \log^3 n/p)$ steps. Thus each leader "discovers" its closest neighbor leader in both directions, and therefore all leaders are marked as being in the same component.

Extending this technique to even 3-regular graphs requires considerably more complicated machinery and the general bound is weaker. (In particular, the walks need to have length $O(n^2 \log^5 n/p^2)$ and we need to try $O(\log n)$ choices for leaders.)

Our main results are:

THEOREM 2.4. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a subset of p vertices chosen at random according to the stationary distribution. Let $S_v(t)$ denote the set of vertices seen in a random walk of length t starting at v. Define the random variable C_p by*

$$C_p = \inf\{t \ : \ \bigcup_{l \in L} S_l(t) = V\},$$

that is, C_p is the time needed for p parallel random walks to visit all the vertices in the graph. Then

$$\mathbf{E}(C_p) = O\left(\frac{m^2 \log^3 n}{p^2}\right).$$

THEOREM 3.1. *There is an algorithm that, given an undirected graph G with n vertices and m edges, and given two vertices s and t of G, decides* USTCON *with one-sided error using space p and time $O(m^2 \log^5 n/p)$. If s and t are in the same connected component, the algorithm outputs YES with probability $1 - O(n^{-1})$, otherwise it outputs NO.*

Remark. The algorithm mentioned in Theorem 3.1 runs in time, that is, within a $\log^5 n$ factor of our target time-bound of $O(mn/p)$ for linear-sized graphs. It is conceivable that a better analysis would lead to a similar algorithm with a lower overhead factor; however, the path example shows that a factor of $\log^3 n$ is inherent to our approach.

2. Covering a graph with p random walks. In this section we derive an upper bound on the time taken by p parallel and independent walks to cover the graph (Theorem 2.4).

We denote by $\{v, w\}$ the undirected edge between vertices v and w and by $[v, w]$ its directed version. For the purposes of the proof, we need to look at the random walk in two ways: first, as a Markov chain $X(t)$ where each state is a vertex in G (the vertex process); second, as a Markov chain $Y(t)$ where each state is a directed edge (the edge process). The transition rule for the vertex process is that if $X(t) = v$, then $X(t+1)$ is equally likely to be any of the neighbors of vertex v. The edge process is defined by $Y(t) = [X(t-1), X(t)]$, $t \geq 1$. The stationary distribution of the vertex process, denoted π, is given by $\pi_v = d_v/(2m)$ where d_v is the degree of the vertex v, and the stationary distribution of the edge process, denoted π', is given by $\pi'_{[v,w]} = 1/(2m)$.

Let $N_v(u, T)$ (respectively, $N_v([u, w], T)$) be the number of visits to the vertex u (respectively, traversals of $[u, w]$) in a random walk of length T starting at v. (By definition, $N_v(v, 0) = 0$.) Let $S_v(T)$ (respectively, $E_v(T)$) be the set of vertices (edges) visited in a random walk of length T starting at v. Finally, let $H_v(u)$ (respectively, $H_v([u, w])$ be the first time the vertex u (the edge $[u, w]$) is encountered by a random walk starting from v. (We define $H_v(v)$ to be $1/\pi_v$, i.e., the return time to v.) For all of these random variables, a replacement of the subscript v with the subscript π (respectively, $[v, w]$) denotes a random walk starting at the stationary distribution (respectively, the directed edge $[v, w]$).

LEMMA 2.1. *Let G be a connected, undirected graph on n vertices. Consider a random walk of length τ starting from the stationary distribution. Then for every directed edge $[v, w]$,*

$$\Pr\Big([v, w] \in E_\pi(\tau)\Big) \geq \frac{\mathbf{E}\Big(N_\pi([v, w], \tau)\Big)}{1 + \mathbf{E}\Big(N_{[v,w]}([v, w], \tau)\Big)}.$$

Proof. Clearly

$$\begin{aligned}\mathbf{E}\Big(N_\pi([v, w], \tau)\Big) &= \textstyle\sum_{1 \leq t \leq \tau} \Pr\Big(H_\pi([v, w]) = t\Big)\Big(1 + \mathbf{E}\Big(N_{[v,w]}([v, w], \tau - t)\Big)\Big) \\ &\leq \Pr\Big(H_\pi([v, w]) \leq \tau\Big)\Big(1 + \mathbf{E}\Big(N_{[v,w]}([v, w], \tau)\Big)\Big).\end{aligned}$$

But $\Pr\Big(H_\pi([v, w]) \leq \tau\Big) = \Pr\Big([v, w] \in E_\pi(\tau)\Big)$, yielding the lemma. □

LEMMA 2.2. *Let G be a connected, undirected graph with n vertices and m edges. Then for every directed edge $[v, w]$,*

$$\mathbf{E}\Big(N_{[v,w]}([v, w], \tau)\Big) \le \frac{\tau}{2m} + \gamma\sqrt{\tau \ln n},$$

where γ is an absolute constant.

Proof. We consider the edge process Y_t. From standard results in renewal theory [9, Thm. 3.7.1] we obtain that

$$\text{(1)} \qquad \mathbf{E}\Big(N_{[v,w]}([v, w], \tau)\Big) = \pi'_{[v,w]}\Big(\tau + \mathbf{E}\Big(H_{Y_{[v,w]}(\tau)}([v, w])\Big)\Big) - 1.$$

Clearly

$$\text{(2)} \qquad \mathbf{E}\Big(H_{Y_{[v,w]}(\tau)}([v, w])\Big) = \mathbf{E}\Big(H_{X_w(\tau)}(v)\Big) + \mathbf{E}\Big(H_v([v, w])\Big).$$

Let $d(x, y)$ be the distance (the length of the shortest path) between two vertices x and y in G. Let c be a sufficiently large constant.

We first bound $\mathbf{E}\Big(H_{X_w(\tau)}(v)\Big)$ using the fact that $d(X_w(\tau), w)$ is not likely to be more than $c\sqrt{\tau \ln n}$, for some $c > 0$ such that $c\sqrt{\tau \ln n}$ is an integer. By the law of total probability

$$\text{(3)} \qquad \begin{aligned} &\mathbf{E}\Big(H_{X_w(\tau)}(v)\Big) = \\ &\mathbf{E}\Big(H_{X_w(\tau)}(v) \mid d(X_w(\tau), v) \le c\sqrt{\tau \ln n}\Big) \Pr\Big(d(X_w(\tau), v) \le c\sqrt{\tau \ln n}\Big) \\ &+ \mathbf{E}\Big(H_{X_w(\tau)}(v) \mid d(X_w(\tau), v) > c\sqrt{\tau \ln n}\Big) \\ &\times \Pr\Big(d(X_w(\tau), v) > c\sqrt{\tau \ln n}\Big). \end{aligned}$$

Since $d(X_w(\tau), v) \le 1 + d(X_w(\tau), w)$, we obtain from the main result of [4] that

$$\text{(4)} \qquad \begin{aligned} \Pr\Big(d(X_w(\tau), v) > c\sqrt{\tau \ln n}\Big) &\le \Pr\Big(d(X_w(\tau), w) \ge c\sqrt{\tau \ln n}\Big) \\ &\le \sum_{x : d(w,x) \ge c\sqrt{\tau \ln n}} 2\left(\frac{\pi_x}{\pi_w}\right)^{\frac{1}{2}} \exp\left(-\frac{d(w, x)^2}{2\tau}\right) \\ &\le 3n^{\frac{1}{2}} \exp\left(-\frac{c^2 \tau \ln n}{2\tau}\right) \le \frac{1}{n^3}, \end{aligned}$$

for a sufficiently large c.

For any two vertices x and y in the same component we can apply the bound implicitly proven in [1]

$$\text{(5)} \qquad \mathbf{E}\Big(H_x(y)\Big) \le 2md(x, y) \le n^3.$$

Plugging equation (5) and equation (4) in equation (3) we obtain that

$$\text{(6)} \qquad \mathbf{E}\Big(H_{X_w(\tau)}(v)\Big) \le 2cm\sqrt{\tau \ln n} + 2.$$

Turning to the second term of the right side of equation (2), we observe that

$$\mathbf{E}\Big(H_v([v,w])\Big) \le 2m+1, \tag{7}$$

because the expected time to return to v given that v was left through an edge other than $[v,w]$ is at most $2m/(d_v-1)$ and the expected number of returns to v before exiting through $[v,w]$ is d_v-1. (The former fact follows from $2m/d_v = \mathbf{E}(H_v(v)) \ge (d_v-1)/d_v \cdot \mathbf{E}(H_v \mid v \text{ not left via } [v,w])$.)

Combining equations (6), (7), and (2), we obtain that

$$\mathbf{E}\Big(H_{Y_{[v,w]}(\tau)}([v,w])\Big) \le 2cm\sqrt{\tau \ln n} + 2m + 3.$$

Finally, from equation (1), because $\pi'_{[v,w]} = 1/(2m)$ for any edge $[v,w]$

$$\mathbf{E}\Big(N_{[v,w]}([v,w],\tau)\Big) \le \frac{\tau}{2m} + c\sqrt{\tau \ln n} + O(1).$$

From here, the lemma follows with an appropriate value for γ. □

LEMMA 2.3. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p vertices (called leaders) in G chosen independently according to the stationary distribution. For every constant $c_1 > 0$ there exists a constant c_2 such that for every directed edge $[v,w]$, a set of p walks of length $c_2 m^2 \ln^3 n/p^2$, one from each of the leaders, satisfies*

$$\Pr\Big([v,w] \in \bigcup_{l \in L} E_l(c_2 m^2 \ln^3 n/p^2)\Big) \ge 1 - \frac{1}{n^{c_1}}.$$

Proof. For $p = O(\log n)$ the conclusion is obvious. For larger p we start from

$$\Pr\Big([v,w] \notin \bigcup_{l \in L} E_l(\tau)\Big) = \prod_{l \in L} \Pr\Big([v,w] \notin E_l(\tau)\Big),$$

and, since each vertex l is chosen independently according to the stationary distribution, Lemma 2.1 gives us a bound on $\Pr([v,w] \notin E_l(\tau))$. By Lemma 2.2 and because $\mathbf{E}\Big(N_\pi([v,w],\tau)\Big) = \tau/2m$, there exists a constant $c_3 > 0$ such that

$$\Pr\Big([v,w] \notin \bigcup_{l \in L} E_l(\tau)\Big) \le \left(1 - \frac{c_3\sqrt{\tau}}{m\sqrt{\ln n}}\right)^p,$$

provided that $\tau = O(m^2 \log n)$. Now taking $\tau = c_2 m^2 \ln^3 n/p^2$ yields the result. □

THEOREM 2.4. *Let $G = (V,E)$ be a connected, undirected graph with n vertices and m edges. Let L be a subset of p vertices chosen at random according to the stationary distribution. Let $S_v(t)$ denote the set of vertices seen in a random walk of length t starting at v. Define the random variable C_p by*

$$C_p = \inf\{t : \bigcup_{l \in L} S_l(t) = V\},$$

that is, C_p is the time needed for p parallel random walks to visit all the vertices in the graph. Then

$$\mathbf{E}(C_p) = O\left(\frac{m^2 \log^3 n}{p^2}\right).$$

Proof. Corollary of Lemma 2.3. □

In fact Lemma 2.3 implies the stronger result that the time needed for p parallel random walks to traverse every edge in the graph is $O\big(m^2 \log^3 n/p^2\big)$.

3. An algorithm for USTCON in $O(p)$ space. We now present the algorithm for USTCON using $O(p)$ space. As a subroutine, we use a standard Union/Find algorithm.

We use three constants, k_1, k_2, and k_3, in the description of the algorithm, which must be chosen sufficiently large. The choice of these constants determines the error probability of the algorithm. For ease of reference, we note here that k_1 is the constant c_2 of Lemma 2.3, Theorem 4.1, and Corollary 4.2, k_2 is the constant c_2 of Lemma 4.6, and k_3 is bound in Theorem 4.7.

```
algorithm STConn;
begin
  do $k_3 \ln n$ times begin
    Let $L$ be a set of $p$ elements of $V$, chosen independently at
      random according to the stationary distribution;
    $L := L \bigcup \{s, t\}$;
    (* Set($l$) are the leaders known to be connected to leader $l$ *)
    for every $l$ in $L$ do Set($l$) := $\{l\}$;
    do $k_2 \ln n$ times begin
      for every $l$ in $L$ do begin
        Take a random walk $X_l(t)$, starting at $l$, of length
          $k_1 m^2 \ln^3 n/p^2$.
        At each step, if $X_l(t) \in L$ then
          Union(Find($X_l(t)$), Find($l$));
      end ;
    end ;
    if Find($s$) =Find($t$)
      then return ("YES: $s$ and $t$ are connected")
  end ;
return ("NO: $s$ and $t$ don't seem to be connected") end .
```

THEOREM 3.1. *Given an undirected graph G with n vertices and m edges, and given two vertices s and t of G, the algorithm* **STConn** *decides* USTCON *with one-sided error using space $O(p)$ and time $O(m^2 \log^5 n/p)$.*

Proof. Choosing a random set of p vertices according to the stationary distribution can be done in $O(m)$ steps using $O(p \log n)$ random bits and $O(p)$ space. Since only $O(p)$ space can be used to store L and do lookups on, a binary search tree or a perfect hash function must be used. (Constructing a perfect hash function for storing L requires expected time $O(p)$ [6].) If the unions are weighted and each union causes path compression on all elements of the set, then each find has cost $O(1)$. Since at most $O(n)$ non-trivial unions are performed, the cost of all the unions is $O(n \log n)$. Performing all $O(\log n)$ random walks of length $O(m^2 \log^3 n/p^2)$ takes time $O(m^2 \log^4 n/p^2)$ per leader for a total of $O(m^2 \log^4 n/p)$ time. Since this is also the total number of finds and lookups performed, this is the running time of each execution of the outermost loop. □

Remark. Note that this algorithm is easily parallelizable using p processors and $O(p)$ space. The parallel hashing scheme described in [7] can be used to implement a parallel version of this algorithm that runs on p processors, $n^\epsilon \leq p \leq n^{1-\epsilon}$, $\epsilon > 0$, that are connected by a bounded degree network. Briefly, storing the leader set using parallel hashing allows for the p processors to execute parallel unions and parallel finds in time $O(p^{\epsilon'})$ for any $\epsilon' > 0$, and consequently the random walks from each of the leaders can be executed in parallel. The resulting parallel implementation of the **STConn** algorithm runs in time $O(m^{2+\epsilon''}/p^2)$ ($\epsilon'' > \epsilon'$).

4. The correctness of STConn. Because our algorithm has one-sided error, it suffices to analyze its correctness in the case when s and t are in the same component of G. If G is actually connected, the results of §2 show that, in one pass through the inner loop of **STConn**, every edge is traversed with high probability. From this, it is possible to deduce that every leader is discovered by some leader. As mentioned earlier, however, this is not enough to prove that s and t become linked. The rest of this section shows that s and t will be "linked up" with high probability after $O(\log n)$ passes through the inner loop.

THEOREM 4.1. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders, each chosen at random according to the stationary distribution. Then for any $c_1 > 0$ there is a constant $c_2 > 0$ such that*

$$\Pr(L \cap S_{[v,w]}(c_2 m^2 \ln^3 n/p^2) \neq \emptyset) \geq 1 - \frac{1}{n^{c_1}},$$

where $S_{[v,w]}(T)$ denotes the set of distinct vertices visited in a T step random walk starting at $[v, w]$.

Proof. The proof is very similar to that of Lemma 2.3. As before the case $p = O(\log n)$ is trivial.

Let e be a directed edge chosen uniformly at random. By a proof virtually identical to that of Lemma 2.1,

$$\Pr(e \in E_{[v,w]}(\tau)) \geq \frac{\mathbf{E}(N_{[v,w]}(e, \tau))}{1 + \mathbf{E}(N_e(e, \tau))}.$$

Obviously, if e is chosen uniformly at random then

$$\mathbf{E}(N_{[v,w]}(e, \tau)) = \frac{\tau}{2m}.$$

By Lemma 2.2

$$\mathbf{E}(N_e(e, \tau)) \leq \frac{\tau}{2m} + \gamma\sqrt{\tau} \ln n.$$

Hence, for e chosen uniformly at random, there exists a constant c_3 such that

$$\Pr(e \in E_{[v,w]}(c_2 m^2 \ln^3 n/p^2)) \geq c_3 \frac{\ln n}{p},$$

provided that $P = \Omega(\log n)$.

In order to choose a leader according to the stationary distribution, one can choose a directed edge e uniformly at random and let the leader be the head of e. Since the probability of reaching a leader is greater than or equal to the probability of traversing the edge chosen to determine it, we obtain that

$$\Pr(L \cap S_{[v,w]}(c_2 m^2 \ln^3 n/p^2) = \emptyset) = (1 - \Pr(e \in E_{[v,w]}(c_2 m^2 \ln^3 n/p^2)))^p \leq \frac{1}{n^{c_1}},$$

for a sufficiently large c_2. □

COROLLARY 4.2. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders chosen at random according to the stationary distribution. Then for any $c_1 > 0$ there is a constant $c_2 > 0$ such that*

$$\Pr(L \cap S_s(c_2 m^2 \ln^3 n/p^2) \neq \emptyset) \geq 1 - \frac{1}{n^{c_1}}$$

and

$$\Pr(L \cap S_t(c_2 m^2 \ln^3 n/p^2) \neq \emptyset) \geq 1 - \frac{1}{n^{c_1}}.$$

Let L be any set of p leaders. We say the set L is *good* if for an absolute constant k_1 (determined in Lemma 4.3 below) the following two properties hold.

Property 1. The probability that a set of p independent random walks of length $\tau = \frac{1}{2}k_1 m^2 \ln^3 n/p^2$, one from each leader in L, traverses every edge in G is at least $1 - 1/n^3$.

Property 2. For every edge $[v, w] \in G$, the probability that a random walk of length τ starting from $[v, w]$ visits some leader in L is at least $1 - 1/n^3$.

LEMMA 4.3. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders chosen uniformly at random according to the stationary distribution. Then* $\Pr(L \text{ is good}) \geq 1 - 2/n$.

Proof. Say that a set of random walks, one from each of the leaders, is unsuccessful for $[v, w]$ if $[v, w]$ is not visited by any of them. Letting $c_1 = 6$ in Lemma 2.3 , we see that at most $1/n^3$ of the possible leader sets can have probability greater than $1/n^3$ of yielding unsuccessful random walks for any fixed $[v, w]$. Similarly, letting $c_1 = 6$ in Theorem 4.1, we see that at most $1/n^3$ of the possible leader sets have probability greater than $1/n^3$ of remaining undiscovered in a random walk of length τ from any fixed edge $[v, w]$. The probability that a leader set is not good is bounded by the sum of the probabilities that it isn't good because it violates properties 1 or 2. Since there are less than $n^2/2$ edges, the probability that a leader set is bad is bounded by $1/n$. The constant k_1 is determined by the requirements of Lemma 2.3 and Theorem 4.1. □

LEMMA 4.4. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders chosen uniformly at random according to the stationary distribution. Suppose that L is a good set of leaders. Let A and B be a partition of L into two nonempty subsets. Consider a random walk of length* 2τ *from each of the leaders in L. Then the probability that some leader in A is visitedfrom some leader in B or vice versa is greater than* 1/18.

Proof. (Unless stated otherwise, all edges referred to in this proof are directed.) We assign to each edge in the graph two labels: a "To" label T and a "From" label F. These labels are subsets of the set $\{A, B\}$. By definition, $A \in T(e)$ (respectively, $B \in T(e)$) if the probability that e is visited by a walk of length τ emanating from one of the leaders in A (respectively, a walk from one of the leaders in B) is at least 1/3. Analogously, $A \in F(e)$ (respectively, $B \in F(e)$) if the probability that some leader in A (respectively, B) is visited in a random walk of length τ starting from e is at least 1/3.

Properties 1 and 2 of good leader sets imply that for each edge neither label is empty. We now consider four cases.

Case 1. There is some edge $[v, w]$ with $A \in F([v, w])$ and $B \in T([v, w])$ or vice versa. Then with probability $\geq$ 1/3 edge $[v, w]$ is visited by one of the random walks of length τ originating in A and with probability $\geq$ 1/3 a leader in B is visited in the remaining at least τ steps. Hence, with probability $\geq$ 1/9 a leader in B is visited from a leader in A. After eliminating this case, the only remaining possibility is that for every edge $F([v, w]) = T([v, w]) = \{A\}$ or $F([v, w]) = T([v, w]) = \{B\}$.

Case 2. There is some undirected edge $\{v, w\}$ such that $F([v, w]) = T([v, w]) = \{A\}$, and $F([w, v]) = T([w, v]) = \{B\}$. Then with probability $\geq$ 1/3, $[v, w]$ is visited by one of the walks of length τ originating in A and hence the vertex v is visited by one of these walks with probability $\geq$ 1/3. Since a leader in B is visited from $[w, v]$ in τ steps with probability $\geq$ 1/3, a leader in B is visited from v in τ steps with probability $\geq$ 1/3. Hence with probability $\geq$ 1/9 a leader in B is visited from a leader in A.

Case 3. No label in the graph contains A or no label in the graph contains B. Without loss of generality, consider the first of the two conditions. Then every edge directed towards leaders in A, has a "To" label of B. Therefore, with probability $\geq 1/3$, each such edge is visited by one of the random walks of length τ originating in B and a leader in A is immediately visited. Hence, with probability $\geq 1/3$, a leader in A is visited from a leader in B.

Case 4. For each undirected edge $\{v, w\}$, we have $T([v, w]) = F([v, w]) = T([w, v]) = F([w, v]) = \{A\}$ or we have $T([v, w]) = F([v, w]) = T([w, v]) = F([w, v]) = \{B\}$. Since case 3 does not hold and the graph is connected, there must be a vertex v that is simultaneously the endpoint of some all-A labeled edge and some all-B labeled edge. Assume without loss of generality that at least 1/2 of the undirected edges with one endpoint at v have all their labels equal to B. Then since some edge $[w, v]$ has an A T-label, with probability $\geq 1/3$ v is visited in the first τ steps of the random walks originating at A. Since the majority of edges leaving v have a B F-label, with probability $\geq 1/2$ one of these edges will be traversed and then with probability $\geq 1/3$, a leader in B will be reached during the remaining at least τ steps. Hence with probability $\geq 1/18$ a leader in B is visited from a leader in A. □

We say that a subset of leaders forms a component if, during some prior phase of the algorithm, they have all been connected up with one another. During a particular phase, we say that a component C is *successful* if it discovers some other component or some other component discovers it. The previous lemma proves that, if the leader set is good, every component has probability at least 1/18 of being successful. The next lemma shows that the number of separate components decreases exponentially with the number of phases.

LEMMA 4.5. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders chosen uniformly at random according to the stationary distribution. Suppose that L is a good leader set. Let N_i be the number of components after the ith phase. Then there exist constants α and β, with $0 < \alpha, \beta < 1$, such that if $N_i > 1$ then*

$$\Pr(N_{i+1} > \beta N_i) \leq \alpha.$$

Proof. Plainly, N_{i+1} equals N_i minus the number of non-redundant links formed in phase i. Since the number of such links formed in phase i exceeds one-half the number of successful components, and the previous lemma shows that the probability that a component is successful is at least 1/18,

$$\mathbf{E}(\text{number of links formed in phase } i) \geq \frac{1}{2 \cdot 18} N_i.$$

Hence,

$$\mathbf{E}(N_{i+1}) \leq (1 - \frac{1}{36})N_i$$

and so there is a positive constant $\beta < 1$ such that

$$\Pr(N_{i+1} > \beta N_i) \leq \alpha. \qquad □$$

LEMMA 4.6. *Let G be a connected, undirected graph with n vertices and m edges. Let L be a set of p leaders chosen uniformly at random according to the stationary distribution. Suppose that L is a good leader set. Let N_i be the number of components after the ith phase. Then for any constant $c_1 > 0$, there is a constant $c_2 > 0$ such that*

$$\Pr(N_{c_2 \ln n} > 1) \leq \frac{1}{n^{c_1}}.$$

Proof. We say that a phase is successful if $N_{i+1} \leq \beta N_i$. Since the leader set is fixed and good, successive phases are independent (the random walks are independent), and by the previous lemma, phase i has probability greater than $1-\alpha$ of being successful for each i. But the probability that $N_{c_2 \ln n}$ is greater than 1 is bounded by the probability that there are fewer than $\ln_{1/\beta} n$ successful phases out of $c_2 \ln n$ phases. This in turn is bound by the probability that there are fewer than $\ln_{1/\beta} n$ successes in $c_2 \ln n$ Bernoulli trials with probability greater than $1-\alpha$ of success, which by Chernoff's bound is less than $1/n^{c_1}$, for appropriately chosen c_2. □

THEOREM 4.7. *The algorithm* **STConn** *decides* USTCON *using space $O(p)$ and time $O((m^2 \log^5 n)/p)$ with one-sided error. If s and t are in the same connected component, the algorithm fails to output YES with probability $O(n^{-1})$; if s and t are in different components, it outputs NO.*

Proof. If the graph consists of a single connected component, then we need only consider one execution of the outer loop of the algorithm, wherein the algorithm can fail to output YES when it should if either the leader set is not good or the leader set is good but the number of components did not reduce to 1. By Lemma 4.3, the former has probability at most $1/n$, and by Lemma 4.6 the latter, when choosing the constant k_3 appropriately, has probability at most $1/n$, and so the theorem follows in this case.

The other case is when s and t are in a single component C containing $\tilde{n}$ vertices and $\tilde{m}$ edges. If $m^2/p^2 > \tilde{m}\tilde{n}$ or $\tilde{n} < \ln^{3/2} n$, then in $k_3 \ln n$ random walks of length $k_1 m^2 \ln^3 n/p^2$ starting from s, the vertex t will be seen with overwhelming probability, since the expected cover time of the component is bounded by $2\tilde{m}\tilde{n}$ [1].

Otherwise, if $m^2/p^2 < \tilde{m}\tilde{n}$, the algorithm can fail to output YES when it should, if either none of the $c_0 \ln n$ selections of leaders include enough leaders that are in the component C, or if some selection of leaders includes enough leaders in C but the associated random walks do not succeed in connecting s to t. For the latter case, we observe that, in each of the $c_0 \ln n$ executions of the outer loop of the algorithm, the expected number of leaders that are chosen from C is $\tilde{p} = p\tilde{m}/m$. If $\tilde{p}/2$ leaders are indeed chosen from C, then since

$$\tau = \frac{c_3 m^2 \ln^3 n}{p^2} = \frac{c_3 \tilde{m}^2 \ln^3 n}{\tilde{p}^2},$$

the analysis given for a single connected graph on $\tilde{n}$ vertices and $\tilde{m}$ edges with $\tilde{p}$ leaders yields a failure probability of $O(\tilde{n}^{-1}) = o(1)$. To bound the probability that a leader selection is not sufficiently dense, we note that the probability that fewer than $\tilde{p}/2$ leaders are chosen from C is bounded by the probability of fewer than $\tilde{p}/2$ successes in p trials with probability $\tilde{m}/m$ of success. By standard bounds, this probability is at most

$$e^{-\frac{1}{8}p\frac{\tilde{m}}{m}} \leq e^{-\frac{1}{8}\sqrt{\frac{\tilde{m}}{\tilde{n}}}} \leq c$$

for some constant $c < 1$ (since $m/p < \sqrt{\tilde{m}\tilde{n}}$). Therefore, the probability that a single execution of the outermost loop fails is bounded by $c + o(1)$, and hence the overall probability of failure is bounded by $O(n^{-1})$, for a sufficiently large constant k_3. □

5. Open problems. Can the bound on the parallel cover time given in Theorem 2.4 be improved? Note that we bound the cover time for all vertices by bounding the cover time for all edges. It is not clear that this is necessary.

Theorem 3.1 shows that for p slightly larger than the average degree m/n, our algorithm runs faster than the random walk. Devising an algorithm that runs in time $O(mn \log^k n/p)$ is perhaps the most interesting open problem.

There is no fundamental reason why our upper bound is the best possible. We thus hope that this work will spark interest in proving a time-space tradeoff for USTCON, even in a restricted model of space-bounded computation such as the JAGs of Cook and Rackoff [5]. Beame et al. [2] give a number of time-space tradeoffs for structured models based on automata that traverse graphs. For one natural variant that admits implementations of our algorithms, they show that the product of time and space is $\Omega(n^2)$ for d-regular graphs ($d \geq 3$), and is $\Omega(mn)$ for non-regular graphs.

Acknowledgments. We are grateful to Lyle Ramshaw for a thorough reading of the manuscript and many useful comments and corrections. We also thank Larry Ruzzo and Martin Tompa for their many comments and suggestions, and also the anonymous referees.

REFERENCES

[1] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff, *Random walks, universal traversal sequences, and the complexity of maze problems*, in 20th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1979, pp. 218–223.

[2] P. Beame, A. Borodin, P. Raghavan, W. L. Ruzzo, and M. Tompa, *Time-space tradeoffs for undirected graph traversal*, in Proc. of the 31st IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, CA, 1991, pp. 429–438.

[3] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa, *Two applications of inductive counting for complementation problems*, SIAM J. Comput., 18 (1989), pp. 559–578. See also 18 (1989), p. 1283.

[4] T. K. Carne, *A transmutation formula for Markov chains*, Bull. Sci. Math., 109 (1985), pp. 399–405.

[5] S. A. Cook and C. W. Rackoff, *Space lower bounds for maze threadability on restricted machines*, SIAM J. Comput., 9 (1980), pp. 636–652.

[6] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with O(1) worst case access time*, J. ACM, 31 (1984), pp. 538–544.

[7] A. R. Karlin and E. Upfal, *Parallel hashing: An efficient implementation of shared memory*, J. ACM, 35 (1988), pp. 876–892.

[8] H. R. Lewis and C. H. Papadimitriou, *Symmetric space-bounded computation*, Theoret. Comput. Sci., 19 (1982), pp. 161–187.

[9] S. Ross, *Stochastic Processes*, Wiley, New York, 1983.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 335–354, April 1994

BOUNDS ON THE COSTS OF MULTIVALUED REGISTER IMPLEMENTATIONS*

SOMA CHAUDHURI[†] AND JENNIFER L. WELCH[‡]

Abstract. A fundamental aspect of any concurrent system is how processes communicate with each other. Ultimately, all communication involves concurrent reads and writes of shared memory cells, or *registers*. The stronger the guarantees provided by a register, the more useful it is to the user, but the harder it may be to implement in practice. This paper considers the problem of implementing a k-ary regular (respectively, safe) register out of binary regular (respectively, safe) registers, assuming a single writer. While algorithms have been developed previously for these problems, no nontrivial lower bounds were known. The cost measures considered here are the number of physical registers and the number of reads and writes on the physical registers required to implement the logical register. Tight bounds are obtained on the cost measures in many cases, and interesting trade-offs between the cost measures are identified. The lower bounds are shown using information-theoretic techniques. Two new algorithms are presented that improve on the costs of previously known algorithms: the *hypercube* algorithm implements a k-ary safe register out of binary safe registers, requiring only one physical write per logical write; and the *tree* algorithm implements a k-ary regular register out of binary regular registers, requiring only $\lceil \log k \rceil$ physical operations per logical operation. Both algorithms use novel combinatorial techniques.

Key words. registers, concurrent distributed system, concurrent computation, shared memory registers, time and space complexity

AMS subject classifications. 68Q22, 68Q25

1. Introduction.

A fundamental aspect of any concurrent system is how processes communicate with each other. Ultimately, all communication involves concurrent accesses to shared memory cells, or registers. The stronger the guarantees provided by the shared memory, the more useful it is to the user, but the harder it may be to implement in practice. Thus it is of interest to determine which types of registers can implement which other types. Many such implementations are known, e.g., [1], [2], [6], [7], [10], [11], [12], [13], [15], [16], [17], among many others.

The contribution of this paper is to study the *costs* of implementing one type of register (the *logical* register) out of registers of another type (the *physical* registers). Cost measures considered are the number of physical registers and the number of operations on the physical registers used to perform the operations of the implemented register. Bounds on the number of physical operations can be used to obtain time bounds for the logical operations in terms of the time taken by the physical operations.

A *register* is a shared variable or memory cell that supports concurrent reading and writing by a collection of processing entities. The operations of reading and writing are not instantaneous; instead, they have duration in time, from a starting point to an ending point. Although each entity accessing a register is assumed to issue operations sequentially, operations on behalf of different entities can overlap in time.

A variety of types of registers can be defined, differing in several dimensions, including the number of concurrent readers supported, the number of concurrent writers supported, the number of values the register can take on, and the strength of the consistency guarantees provided in the presence of concurrent operations. Throughout this paper we assume there

*Received by the editors November 19, 1990; accepted for publication (in revised form) November 30, 1992. This work was done while the authors were at the Department of Computer Science, University of North Carolina at Chapel Hill, and supported in part by National Science Foundation grant CCR-9010730, an IBM Faculty Development Award, and National Science Foundation Presidential Young Investigator Award CCR-9158478.

[†] Department of Computer Science, Iowa State University, Ames, Iowa 50011 (chaudhur@cs.iastate.edu).

[‡] Department of Computer Science, Texas A&M University, College Station, Texas 77843 (welch@cs.tamu.edu).

is only one writer, leaving three parameters of interest: the number of readers, the number of values, and the consistency guarantees. We distinguish between 1-reader registers and n-reader registers, for $n > 1$, and between binary registers and k-ary registers, for $k > 2$. (A k-ary register can take on k different values.)

Lamport [6] defines three consistency guarantees of increasing strength, namely, safe, regular, and atomic. Roughly speaking, a read of a *safe* register always returns the most recent value written to the register, unless the read overlaps with a write, in which case any legal value of the register can be returned. A read of a *regular* register always returns the most recent value written, unless the read overlaps one or more writes, in which case it returns either the old value or one of the values written by an overlapping write. An *atomic* register provides the illusion, via the values returned by read operations, that each operation happens at a single instant in time within its range, i.e., that the operations are totally ordered. In this paper, we only consider safe and regular registers. In particular, we consider the problem of implementing an n-reader k-ary regular (respectively, safe) register out of n-reader binary regular (respectively, safe) registers.

We study the costs incurred by these implementations. Let M, R, and W be the minima, over all implementations between two particular types of registers, of the number of physical registers, the maximum number of physical operations in a logical read, and the maximum number of physical operations in a logical write, respectively. Our algorithms will involve no physical reads in a logical write and no physical writes in a logical read. Our lower bound results give bounds on the number of physical reads per logical read, and the number of physical writes per logical write. These are stronger results than just giving bounds on the number of physical operations per logical action.

Our results are summarized in Tables 1 and 2. Table 1 gives the bounds when all algorithms are considered. Table 2 gives the bounds when certain classes of algorithms are considered, as specified by the column labeled S—namely, 1-write algorithms, c-write algorithms, and $\lceil \log k \rceil$-register algorithms. (All logarithms are base 2.)

For implementing a k-ary safe register out of binary safe registers, we show tight bounds of $R = \lceil \log k \rceil$, $W = 1$, and $M = \lceil \log k \rceil$. The upper bound of 1 on W is obtained from a new algorithm, which we call the *hypercube* algorithm. The best previous upper bound on W was $\lceil \log k \rceil$ [6]. These three optimal bounds are not obtained simultaneously in a single algorithm, and in fact, we show some nontrivial trade-offs between the three cost measures.

For implementing a k-ary regular register out of binary regular registers, we show the tight bound that $R = \lceil \log k \rceil$, and the bounds $1 \leq W \leq \lceil \log k \rceil$, and $\max\{\lceil \log k \rceil + 1, 2(\log k) - \log\log k - 2\} \leq M \leq \min\{k - 1, n(3 \log k + 68)\}$, where n is the number of readers of the logical register. The upper bounds on R and W are simultaneously achieved by a new algorithm, which we call the *tree* algorithm. We also present some lower bounds on R and M that follow if we restrict attention to implementations that use only a small constant number of physical writes per logical write.

The lower bounds in Table 1 for safe registers and those on R and W for regular registers are obvious from information-theoretic considerations. All of the remaining lower bounds are new. Little previous work has been done concerning lower bounds or trade-offs for register implementations. One such previous result is in [6], where it is shown that in any implementation of an atomic register using regular registers, a read of the logical register must involve a write to a physical register. Tromp [13] uses this result to show that three binary safe registers are necessary to construct a binary atomic register.

In §2 we present our model and some results for all implementations. Section 3 considers safe registers and §4 considers regular registers. We conclude in §5 with some open questions.

2. Preliminaries. In this section, we give formal definitions for the types of registers that we will study (n-reader, k-ary, safe, and regular), describe the rules we impose on implementing

TABLE 1
Independent bounds for binary to k-ary algorithms.

	Safe		*Regular*	
	lower	*upper*	*lower*	*upper*
R	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\lceil \log k \rceil$
W	1	1	1	$\lceil \log k \rceil$
M	$\lceil \log k \rceil$	$\lceil \log k \rceil$	$\max\{\lceil \log k \rceil + 1,$ $\lceil 2 \log k - \log \log k \rceil - 2\}$	$\min\{k - 1,$ $n(3 \log k + 68)\}$

TABLE 2
Trade-off results for binary to k-ary algorithms.

S		*Safe*		*Regular*	
		lower	*upper*	*lower*	*upper*
$\{A \mid W_A = 1\}$	R_S	$k - 1$	$2^{\lceil \log k \rceil - 1}$	$k - 1$	∞
	M_S	$k - 1$ or k[1]	$2^{\lceil \log k \rceil - 1}$	k	∞
$\{A \mid W_A = c\}$	R_S	$(c!k/2)^{1/c}$	$c - 2 + \lceil k/2^{c-2} \rceil$	$(c!k/2)^{1/c}$	∞
	M_S	$(c!k/2)^{1/c}$	$c - 2 + \lceil k/2^{c-2} \rceil$	$(c!k/2)^{1/c}$	∞
$\{A \mid M_A = \lceil \log k \rceil\}$	W_S	$\lceil \log k \rceil$	$\lceil \log k \rceil$	∞	∞

one type of register with another, and define the cost measures we will use. Then we present some definitions and lemmas that are true for implementations between any types of registers.

2.1. Model. We use a simplified form of the I/O automaton model [9] to describe our system.

To implement a logical register with value set V, where $|V| = k$, we compose a collection of physical registers X_j, $1 \le j \le m$, each with value set $\{0, 1\}$, a collection of read processes RP_i, $1 \le i \le n$, and a single write process WP. The read and write processes implement the protocols used by the readers and writer of the logical register. Each such protocol consists of accessing certain of the physical registers and doing some local computation.

Communication between these components takes place via **actions**. Each action is an output of one component (the component that generates it) and an input to another component. Components are modeled as state machines in which actions trigger transitions. Components have no control over when inputs occur and thus must have a transition for every input in every state. Components do have control over when outputs occur; if an output labels a transition from a state, then the output is **enabled** in that state.

An **execution** of the implementation consists of a sequence in which state tuples (one entry for the state of each component) and actions alternate, beginning with a tuple of initial states. For each action π in the execution, π must be enabled in the preceding state of the component for which it is an output. In the following state tuple, the states of the two components for which π is an input and an output must change according to the transition functions, while the remaining components' states are unchanged.

A **schedule** is the sequence of actions in an execution.

The **logical actions** are READ(i), RETURN(i, v), WRITE(v), and ACK, $1 \le i \le n$ and $v \in V$. READ(i) is an input to RP_i from the outside world; RETURN(i, v) is an output from RP_i to the outside world. WRITE(v) is an input to WP from the outside world and ACK is an output from WP to the outside world. Although we do not explicitly model the outside world with a component, we do assume that for each i, the outside world and RP_i cooperate so that READs and RETURNs strictly alternate, beginning with a READ, and analogously for WP.

[1] $k - 1$ if k is a power of 2; k otherwise.

The **physical actions** are $\text{read}_j(i)$, $\text{return}_j(i, v)$, $\text{write}_j(v)$, and ack_j. The subscript j is between 1 and m; it indicates that X_j is the physical register being read or written. The parameter v is either 0 or 1 and indicates the value being read from or written to X_j. For a fixed j, the parameter i for reads and returns ranges over some subset of $\{0, \ldots, n\}$ of size at most n; this subset indicates which of the read and write processes read X_j. (The value 0 indicates WP.) For a fixed j, there is no parameter i for writes and acks, since there is a unique read or write process that writes X_j.

A READ(i) and its following RETURN(i, v) form a **logical operation**, as do a WRITE(v) and its following ACK. **Physical operations** are defined analogously. An operation is **pending** if its first half is present but not its second half.

We assume that the read and write processes cooperate with the physical registers so that for each i, $0 \leq i \leq n$, and each j, $1 \leq j \leq m$, $\text{read}_j(i)$ and $\text{return}_j(i, *)$ alternate beginning with a read, and analogously for writes. We also assume that no read or write process has a physical operation pending unless it has a logical operation pending.

Each physical register X_j satisfies this liveness property: Immediately after an input action occurs, the matching output is enabled.

A **safe** physical register satisfies the **Safe Property**: For every physical read operation that does not overlap a physical write operation, the value returned is the value written by the most recent physical write operation. If there is no preceding write operation, then it returns the initial value.

A **regular** physical register satisfies the **Regular Property**: Every physical read operation returns a value written by an overlapping write operation or by the most recent preceding write (or the initial value if there is no preceding write).

The read and write processes must work together to implement a logical register. The liveness property for a logical register differs from that for a physical register, as discussed below. A safe (respectively, regular) logical register satisfies the safe (respectively, regular) property, as defined for physical registers, replacing "physical" with "logical."

The liveness property for a logical register is that the implementation must be **wait-free**, meaning that in every finite execution, if a logical operation by RP_i (respectively, WP) is pending, then there is a finite sequence of actions involving only RP_i (respectively, WP) that finishes the operation. Our algorithms actually provide a bounded number of actions, while our lower bounds hold for algorithms satisfying the weaker definition.

A natural question that may arise is why the liveness property is different for physical and logical registers. The wait-free definition for the logical register implies that every logical operation must complete using only physical operations initiated by that logical operation. In the case of the physical register, where we don't model the "internal" actions, this wait-free property reduces to the physical liveness property given.

We now define the cost measures.

Consider two register types, physical and logical, and let A be an algorithm for a physical-to-logical register implementation. Let M_A be the number of physical registers used in A, let R_A be the maximum number of physical operations performed during any logical READ in any execution of A, and let W_A be the maximum number of physical operations performed during any logical WRITE in any execution of A. Given a set S of physical-to-logical register implementations, let M_S be the minimum of M_A over all $A \in S$, R_S be the minimum of R_A over all $A \in S$, and W_S be the minimum of W_A over all $A \in S$. (If $S = \emptyset$, then M_S, R_S, and W_S are infinity.) Finally, let $M = M_S$, $R = R_S$, and $W = W_S$, where S is the set of all physical-to-logical register implementations (for these two types). (The physical and logical register types are implicit parameters to M, R, and W.)

In the rest of this paper, we derive upper and lower bounds on M, R, and W, and trade-offs between them, for different physical and logical register types.

These bounds on R and W can be converted into time bounds for performing logical operations as follows. Suppose we know bounds R_l, R_u, W_l, and W_u such that $R_l \leq R \leq R_u$ and $W_l \leq W \leq W_u$. Let r be an upper bound on the time to read a physical register and let w be an upper bound on the time to write a physical register. Let s be an upper bound on the time for a read or write process to perform an action once it becomes enabled. Our upper bounds on R and W come from algorithms, all of which have the property that no logical READ involves a physical write and no logical WRITE involves a physical read. Since we assume that all physical operations are enclosed within logical operations and that only one physical operation can be pending at a time, we deduce that an upper bound on the worst case time to perform a READ of a logical register that is implemented with physical registers is $R_u(r + s) + s$. Similarly, an upper bound on the worst case time to perform a WRITE of a logical register that is implemented with physical registers is $W_u(w + s) + s$. Our lower bounds on R and W do not assume that logical READs do not involve physical writes, or that logical WRITEs do not involve physical reads, and thus they imply analogous lower bounds on the worst case times.

2.2. General results. Given a finite schedule σ of an algorithm A, let the **configuration** of σ be the tuple of sets of "possible values" of the physical registers at the end of the schedule, i.e., if X_i is the ith physical register, then the ith element of the configuration is the set of all values that could be returned by a physical read of that register at that point, according to the safe/regular property. A configuration is **stable** if each element of the tuple is a singleton set. Thus it can be represented as $x_1 \ldots x_m$, where x_i is the value of register X_i for all i. The **initial configuration** is the (stable) configuration of the empty schedule, consisting of the initial value of each physical register.

Let $\mathcal{WO}$ (for "write-only") be the set of all schedules of A in which only WP takes steps and no physical write is pending. Let $\mathcal{S} = \{C : C$ is the configuration of some $\sigma \in \mathcal{WO}\}$. It is easy to see that all configurations in $\mathcal{S}$ are stable.

For each i, define $L_i : \mathcal{S} \to V$ as follows. $L_i(C)$ is the logical value returned by RP$_i$ when RP$_i$ starts in its local initial state, the physical registers have the values specified in C, and no other read or write process takes a step.

What we would like is a function that returns the value of the logical register when the physical registers are in a given configuration. However, an arbitrary algorithm may have different protocols for different read processes (necessitating our use of a subscript on L), and it may use the history of the read process to determine what value is returned. Thus it might be the case that RP$_i$ returns different values at different times in an execution, even given the same configuration. In order to accommodate such algorithms, we define each L_i specifically when RP$_i$ has taken no steps yet.

The next lemma states that L_i is well defined, i.e., that the current configuration (values of the physical registers) and nothing else determines the value of the logical register (as perceived by RP$_i$). This can be shown by a simple induction on the length of the execution.

LEMMA 2.1. *For any algorithm A, the function L_i is well defined for all i.*

Let $\mathcal{WOC}$ (for "write-only, completed") be the set of all schedules of A in which only WP takes steps and no *logical* WRITE is pending. Let $\mathcal{T} = \{C : C$ is the configuration of some $\sigma \in \mathcal{WOC}\}$. It is easy to see that $\mathcal{T} \subseteq \mathcal{S}$. Every configuration in $\mathcal{T}$ is defined to be a **terminal** configuration.

The next lemma states that if no read process has taken any steps and no logical WRITE is in progress, then each L_i is equal to the value of the most recent WRITE to the logical register.

LEMMA 2.2. *For any algorithm A, if σ is in $\mathcal{WOC}$ with configuration C, then, for all i, $L_i(C)$ equals the value of the most recent* WRITE *(the initial value if σ is empty).*

The previous lemma implies that all the L_i's must agree whenever the argument is in the set $\mathcal{T}$. Thus we define $L : \mathcal{T} \to V$ to be $L(C) = L_i(C)$ for any i. It is easy to see that for each $v \in V$, there is a $C \in \mathcal{T}$ such that $L(C) = v$.

In most of our proofs, we only need to consider situations in which no logical WRITE is pending, and thus we can use the notation L. However, in a few places (notably Lemmas 4.4 and 4.5), we must consider what happens in the middle of a logical WRITE, and thus we must use a specific L_i (we choose L_1 for concreteness).

3. k-ary safe register from binary safe registers. We consider the problem of implementing an n-reader, k-ary, safe register out of n-reader, binary, safe registers, for any $n \geq 1$, where $k > 2$. Subsection 3.1 is devoted to proving tight, independent bounds on R, W, and M. In §3.2, we present an algorithm A such that $W_A = 1$. We also show some nice combinatorial properties related to one-write algorithms. Subsection 3.3 discusses algorithms that allow c physical accesses per logical WRITE. We also give some additional trade-offs between the cost measures.

Let the value set of the logical register be $V = \{0, \ldots, k-1\}$ with initial value $v_0 \in V$.

3.1. Independent bounds. The following theorem gives matching upper and lower bounds on R, W, and M.

THEOREM 3.1. *The implementation of an n-reader, k-ary, safe register by n-reader, binary, safe registers gives the following independent bounds*: $R = \lceil \log k \rceil$, $W = 1$, *and* $M = \lceil \log k \rceil$.

Proof. The upper bounds on R and M follow from the binary representation algorithm in [6] described below. The upper bound on W follows from our hypercube algorithm presented in §3.2. The lower bounds on W and M are obvious.

We now show the lower bound on R. For each $v \in V$, there is a schedule σ_v of A of the form

$$\text{WRITE}(v)\ \alpha_v\ \text{ACK READ}(1)\ \beta_v\ \text{RETURN}(1, v),$$

where α_v consists solely of actions of WP and contains no ACK, and β_v consists solely of actions of RP_1 and contains no RETURN.

By the definition of read processes, for all distinct v and w, $\beta_v \neq \beta_w$ and the maximal common prefix of β_v and β_w is immediately followed by a return(0) action from some physical register X in β_v and by a return(1) action from X in β_w (or vice versa); that is to say, RP_1 does the same thing in β_v and β_w until it reads a different value. Let γ_v be the sequence of physical values read in β_v, for all v.

Thus, if $v \neq w$, then the sequence γ_v of physical values read in β_v is not equal to the sequence γ_w of physical values read in β_w. There are k distinct sequences of physical values corresponding to the γ_v's, i.e., k binary strings. Thus at least one string, say that corresponding to γ_v, must have length at least $\lceil \log k \rceil$, implying that β_v contains at least $\lceil \log k \rceil$ physical reads. □

The **binary representation algorithm** in [6] implements an n-reader, k-ary, safe register out of $\lceil \log k \rceil$ n-reader, binary, safe registers. The write process writes the binary representation of the logical value into the physical registers. Each read process reads all the physical registers and returns the logical value whose binary representation was read, as long as the value is less than k. Otherwise, it returns any value less than k. This algorithm implies that $R \leq \lceil \log k \rceil$, $W \leq \lceil \log k \rceil$, and $M \leq \lceil \log k \rceil$. By Theorem 3.1, the number of registers and number of physical reads in the binary representation algorithm are both optimal.

The **unary representation algorithm** presented next shows that $W \leq 2$. There are $k-1$ physical registers, $X_1, \ldots, X_{k-1}$. Logical value 0 is represented when all registers are 0.

Logical value $v \neq 0$ is represented when X_v is 1 and the other registers are 0. Each read process reads registers X_1, X_2, and so on, in order, until reading a 1, and RETURNs logical value v, where X_v is the register that returned 1. If no register returns 1, then 0 is RETURNed. To WRITE logical value v, assuming w is the old value of the logical register, the write process writes 0 to X_w if $w \neq 0$, and writes 1 to X_v if $v \neq 0$.

In the next subsection, we will present an algorithm which brings down the number of physical writes per logical WRITE to 1.

3.2. One-write algorithms. In this subsection, we discuss the class of one-write algorithms. We show that their existence depends on satisfying a combinatorial coloring property of hypercubes.

Figure 1 presents our new **hypercube algorithm**, which shows that $W \leq 1$; it relies on a function f, which will be defined shortly. For now, assume that k is a power of 2. Later we will show how to remove this restriction.

```
Physical Registers: $X_1, \ldots, X_{k-1}$, initially $f(X_1 \ldots X_{k-1}) = v_0$ and $X_j = 1$ for at most one $j$
Read Process RP$_i$, $1 \leq i \leq n$: variables $x_1, \ldots, x_{k-1}$
        READ($i$):
                for $j := 1$ to $k - 1$ do $x_j :=$ read $X_j$ endfor
        RETURN($i$, $f(x_1 \ldots x_{k-1})$)
Write process WP: variables $x_1, \ldots, x_{k-1}$, initially $x_j$ equals the initial value of $X_j$ for all $j$
        WRITE($v$):
                if $v \neq f(x_1 \ldots x_{k-1})$ then
                        write $\overline{x_j}$ to $X_j$, where $j$ is such that $f(x_1 \ldots x_{j-1}\overline{x_j}x_{j+1} \ldots x_{k-1}) = v$
                        $x_j := \overline{x_j}$
                endif
        ACK
```

FIG. 1. *Hypercube algorithm.*

We notice an interesting relationship between the correctness of the hypercube algorithm and coloring the nodes of a $(k - 1)$-dimensional hypercube with k colors such that each node has a neighbor with each of the $k - 1$ colors other than its own. The following definition and lemmas formalize this idea. (Nodes are labeled with $(k - 1)$-bit strings, the colors are elements of V, and the function is the coloring.)

A function g is said to have the **rainbow-coloring property** if $g : \{0, 1\}^{k-1} \rightarrow V$ such that for all $x \in \{0, 1\}^{k-1}$, and for all $v \in V$, if $v \neq g(x)$, then there exists $y \in \{0, 1\}^{k-1}$ such that $v = g(y)$ and x and y differ in exactly one bit. That is, every node x has a neighboring node with every color other than x's color.

Lemma 3.2 states that if the function f used in the algorithm has the rainbow-coloring property, the hypercube algorithm correctly implements a k-ary safe register using binary safe registers such that each logical WRITE requires one physical write. The rainbow-coloring property ensures that each READ RETURNs the correct value if it does not overlap a WRITE.

LEMMA 3.2. *If function f has the rainbow-coloring property, then the hypercube algorithm is correct.*

We define a function $f : \{0, 1\}^{k-1} \rightarrow V$ for use in the algorithm. Lemma 3.3 shows that f has the rainbow-coloring property. For positive integer $i < k$, let $bin(i)$ be the binary representation of i in $\log k$ bits (remember that k is a power of 2). For $x \in \{0, 1\}^{k-1}$, let x_i be the ith bit of x, i.e., $x = x_1 x_2 \ldots x_{k-1}$. For all $x \in \{0, 1\}^{k-1}$, we define $f(x)$ to be the element of V whose binary representation is $\bigoplus_{x_i=1} bin(i)$, where $\oplus$ represents exclusive-or. This expression consists of $\log k$ bits and thus represents a value in the range 0 to $k - 1$, i.e., a value in V.

The intuition behind the coloring function f is that we want to go from a $(k - 1)$-bit string, the label of a node in the hypercube, to a $(\log k)$-bit string, indicating one of k colors. Given a node with label x, the color assigned is the one whose binary representation is equal to the exclusive-or of the set of $bin(i)$, for all i such that $x_i = 1$. Note that if two nodes x and y differ in the single bit i, then $f(x) \oplus f(y) = bin(i)$. So, given the color of a node x, we can derive the color of any adjacent node y in a consistent manner.

Given this definition of f, the initial values of the physical registers are all 0, except that if $v_0 \neq 0$ then $X_{v_0} = 1$. The computation of j in the writer's code is $bin(j) = bin(v) \oplus bin(f(x_1 \ldots x_{k-1}))$.

LEMMA 3.3. *The function f defined for the hypercube algorithm (when k is a power of 2) has the rainbow-coloring property.*

Proof. First we must show that for all $x, y \in \{0, 1\}^{k-1}$ that differ in exactly one bit, $f(x) \neq f(y)$. Suppose x and y differ in bit i. Then $f(x) \oplus f(y) = bin(i)$. Since $bin(i) \neq 0^{\log k}$, we are done. Second we must show that for all $x, y, z \in \{0, 1\}^{k-1}$ such that $y \neq z$ and y and z both differ from x in exactly one bit, $f(y) \neq f(z)$. This can be shown similarly. These two facts together show that f has the rainbow-coloring property. □

Figure 2 illustrates how our algorithm works in the simple case where $k = 4$. Our hypercube is then a three-dimensional cube, whose vertices can be colored with four colors, r, b, g, and y. Note that the coloring satisfies the rainbow-coloring property.

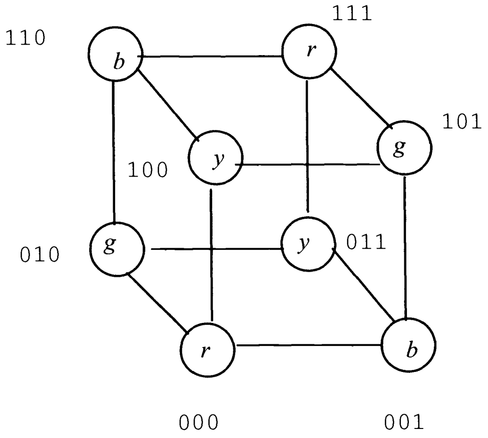


FIG. 2. *An example illustrating the hypercube algorithm.*

Combining Lemmas 3.2 and 3.3 shows that the hypercube algorithm is a one-write algorithm (using $k - 1$ registers) if k is a power of 2. To obtain a one-write algorithm for values of k that are not powers of 2, we modify the power-of-2 hypercube algorithm for $m - 1$ physical registers, where $m = 2^{\lceil \log k \rceil}$, i.e., m is the smallest power of 2 larger than k. The modification is to change the RETURN statement to be RETURN(min$\{k - 1, f(x_1 \ldots x_{m-1})\}$). This implementation of a k-ary register by binary registers will not cause the binary registers to take on all possible 2^{m-1} values, i.e., no stable configuration of the algorithm will be mapped to a value that is out of the range of the logical register. However, a slow read process, which overlaps a number of writes, might (spuriously) observe a configuration corresponding to a value larger than $k - 1$, thus necessitating the modification. Thus we have shown the following theorem.

THEOREM 3.4. *The hypercube algorithm correctly implements a k-ary safe register using binary safe registers.*

The following theorem summarizes our results for the class of 1-write algorithms.

THEOREM 3.5. *Let S be the set of algorithms A such that $W_A \leq 1$. Then*

(i) $k - 1 \leq R_S \leq 2^{\lceil \log k \rceil} - 1$,

(ii) $M_S = k - 1$, *if k is a power of* 2, *and*

(iii) $k \leq M_S \leq 2^{\lceil \log k \rceil} - 1$, *if k is not a power of* 2.

Proof. All the upper bounds follow from the hypercube algorithm. The rest of the proof concerns the lower bounds.

Choose an algorithm $A \in S$. Let C_{v_0} be the initial configuration. For all $v \neq v_0$, let C_v be the configuration of a schedule in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Since α_v only contains one physical write, C_{v_0} and C_v differ in a single bit, say that for physical register X_v.

Since there are $k - 1$ choices for $v \neq v_0$, there are at least $k - 1$ physical registers. Since A was chosen arbitrarily, $M_S \geq k - 1$. The improved lower bound of k for M_S when k is not a power of 2 follows from Lemmas 3.6 and 3.7.

To show $R_S \geq k - 1$, we assume, for contradiction, that $R_A < k - 1$. Consider the schedule READ(1) β RETURN(1, v_0), where β consists solely of actions of RP$_1$ and contains no RETURN. β contains a sequence of less than $k - 1$ physical reads. Let X_v (as defined above) be one of the physical registers not read in β; note that $v \neq v_0$. Since C_{v_0} differs from C_v in the value of register X_v and nowhere else, an easy induction on the length of β shows that WRITE(v) α_v ACK READ(1) β RETURN(1, v_0) is a schedule of A, violating the safe property since $v \neq v_0$. We therefore have a contradiction, implying $R_A \geq k - 1$. □

We now consider the number of registers when k is not a power of 2. Lemma 3.6, which is the converse of Lemma 3.2, shows that the existence of a function with the rainbow-coloring property is necessary for the existence of a one-write algorithm using $k - 1$ registers. Lemma 3.7, which is the converse of Lemma 3.3, shows that when k is not a power of 2, no function with the rainbow-coloring property can exist. Together, these two lemmas imply that if k is not a power of 2, then any one-write algorithm must use more than $k - 1$ registers.

LEMMA 3.6. *If there is an algorithm A with $W_A = 1$ and $M_A = k - 1$, then there exists a function with the rainbow-coloring property.*

Proof. We show that L has the rainbow-coloring property. Recall that L maps $\mathcal{T}$, the set of terminal configurations, to V. We know $\mathcal{T}$ is not empty. Choose any configuration $C \in \mathcal{T}$. Let v be the color of C under L. Every neighbor of C is also in $\mathcal{T}$ and has a unique color different from C's color, since there are only $k - 1$ registers and $k - 1$ possibilities for the next value not to be v. To finish the proof, we note that $\mathcal{T} = \{0, 1\}^{k-1}$, since every neighbor of a terminal configuration is also a terminal configuration. □

LEMMA 3.7. *If k is not a power of* 2, *then there is no function with the rainbow-coloring property.*

Proof. Assume in contradiction that there is a function f with the rainbow-coloring property. Choose any color, say blue, and let b be the number of nodes colored blue by f. Let B be the set of edges in the hypercube that have one endpoint colored blue and one endpoint not colored blue. Since each nonblue node is adjacent to exactly one blue node and there are $2^{k-1} - b$ nonblue nodes, $|B|$ must be $2^{k-1} - b$. However, since each blue node is adjacent to $k - 1$ nonblue nodes and there are b blue nodes, $|B|$ must be $b(k - 1)$. The implication is that $2^{k-1} - b = b(k - 1)$, implying $2^{k-1} = kb$. This implies that k divides 2^{k-1}, contradicting the fact that k is not a power of 2. □

In this subsection, we showed that the existence of a one-write implementation of a k-ary safe register was based on solving an underlying combinatorial problem. Specifically, a one-write algorithm using $k-1$ physical registers exists if and only if we can color a $(k-1)$-dimensional hypercube with k colors such that each node has a neighbor with every color other than its own. We can generalize this to any number of physical registers as follows. A one-write algorithm using m registers exists if and only if we can *partially* color an m-dimensional hypercube with k colors, such that each *colored* node has a neighbor with every color other than its own. By a partial coloring, we mean a coloring where not all nodes of the graph need to be colored.

Lemmas 3.3 and 3.7 imply that there is rainbow-coloring of the $(k-1)$-dimensional hypercube if and only if k is a power of 2. Kant and van Leeuwen [5] have independently shown the same result. Their proof uses notions from coding theory and is based on showing a correspondence between 1-error-correcting codes and these colorings. They applied this result to the file distribution problem.

3.3. c-write algorithms and trade-off results. As we showed for 1-write algorithms, the problem of implementing c-write algorithms can also be shown to have a corresponding parallel in a combinatorial problem. Here, we are interested in a partial coloring of the m-dimensional hypercube such that for each colored node, there exists a node of every other color *within a distance of c* from this node.

This combinatorial characterization also helps us obtain lower bounds for M and R. For example, we know that for there to exist a one-write algorithm that uses m physical registers, there must be a configuration C_0 that differs from $k-1$ different configurations C_v in exactly one bit. Since each configuration is represented in m bits, this says that there is a binary string of length m that differs from $k-1$ different strings of the same length in exactly one bit. To satisfy this combinatorial property, we require that $m \geq k-1$. This sequence of reasoning was implicit in the proof of Theorem 3.5.

Along similar lines, for there to exist a c-write algorithm that uses m registers, there must exist a binary string of length m which differs from $k-1$ different strings of the same length in *at most c* bits.

We formalize this property in Lemma 3.8 and Theorems 3.10 and 3.11. These theorems give lower bounds on M and R for c-write algorithms, i.e., algorithms that use a small bounded number of physical writes per logical WRITE.

The next result is Theorem 3.12, which gives trade-offs on W versus R and M. An application of this result is to give upper bounds on M and R for c-write algorithms.

The final result in this subsection, Theorem 3.13, states that if no more than $\lceil \log k \rceil$ registers are used, then some WRITE must write at least $\lceil \log k \rceil$ physical registers.

LEMMA 3.8. *Given any binary string x of length m, if there are at least k distinct strings of length m which differ from x in at most c bits, where $c \leq (\log k)/3$, then $m \geq (c!k/2)^{1/c}$.*

Proof. Let x be a string of length m. The number of distinct strings of length m that differ from x in at most c bits is

$$\binom{m}{0} + \binom{m}{1} + \binom{m}{2} + \cdots + \binom{m}{c}.$$

Since we know that there are at least k such distinct strings, we have $\sum_{i=0}^{c} \binom{m}{i} \geq k$. We obtain the following upper bound on $\binom{m}{i}$, for all i: $\binom{m}{i} = \frac{m(m-1)(m-2)\cdots(m-i+1)}{i!} \leq m^i/i!$. To get an upper bound on the entire summation, we need the following claim, which is taken from [14]. First, we introduce some notation. Let $s_{m,j}$ denote $\sum_{i=0}^{j} \binom{m}{i}$. Let $b_{m,i}$ denote $\binom{m}{i}$.

CLAIM 3.9. *If $1 \leq j \leq m/3$, then $s_{m,j} \leq 2b_{m,j}$.*

Proof. We compute a lower bound for $b_{m,j}/b_{m,j-1} = \frac{m-j+1}{j}$. Note that $\frac{m-j+1}{j}$ is larger than 2 for $j \leq m/3$. Therefore, for $j \leq m/3$, $b_{m,j}/b_{m,j-1} > 2$. The remaining proof is by induction.

Inductive hypothesis: $s_{m,j} \leq 2b_{m,j}$ for $j \leq m/3$.

Basis: For $j = 1$ (assume $m \geq 3$), $s_{m,0} = b_{m,0} = 1$ and $b_{m,1} = m$. Therefore, $s_{m,1} = m + 1$ and $s_{m,1} \leq 2b_{m,1}$.

Inductive step: Let the inductive hypothesis hold for all l such that $l < j \leq m/3$. We show that it holds for j. By the inductive hypothesis, $s_{m,j-1} \leq 2b_{m,j-1}$. Note that $s_{m,j} = s_{m,j-1} + b_{m,j}$. This implies that $s_{m,j} \leq 2b_{m,j-1} + b_{m,j}$. Also, we showed earlier that $2b_{m,j-1} \leq b_{m,j}$. Therefore, $s_{m,j} \leq b_{m,j} + b_{m,j} = 2b_{m,j}$. □

The above claim holds for $j = c$ since we know that $m \geq \log k$ (it takes $\log k$ bits to represent k distinct values), and this implies that $c \leq m/3$. Now, using the above claim and our previous upper bound for $\binom{m}{i}$, we have $\sum_{i=0}^{c} \binom{m}{i} \leq 2\binom{m}{c} \leq 2m^c/c!$. So, $k \leq 2m^c/c!$; by manipulating this inequality, we get $m \geq (c!k/2)^{1/c}$. □

THEOREM 3.10. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $M_A \geq (c!k/2)^{1/c}$.*

Proof. Given an algorithm A such that $W_A = c$, where $c \leq (\log k)/3$, let C_{v_0} be the initial configuration. Then $L(C_{v_0}) = v_0$. For all $v \neq v_0$, the schedule σ_v of the form WRITE(v) α_v ACK yields the terminal configuration C_v. Since each WRITE can initiate at most c physical writes, each C_v differs in at most c bits from C_{v_0}.

Since there are k values v, there must be at least k terminal configurations C_v differing in at most c bits from C_{v_0}. The number of registers used in the algorithm is M_A. Each terminal configuration is therefore a binary string of length M_A. Therefore, there are at least k strings of length M_A which differ in at most c bits from C_{v_0}. Since, $c \leq (\log k)/3$, Lemma 3.8 applies, and we have the result $M_A \geq (c!k/2)^{1/c}$. □

THEOREM 3.11. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $R_A \geq (c!k/2)^{1/c}$.*

Proof. For any algorithm A, where $W_A \leq c$, consider the following schedules, for all v,

$$\text{WRITE}(v)\ \alpha_v\ \text{ACK READ}(1)\ \beta_v\ \text{RETURN}(1, v),$$

where α_v and β_v contain only physical actions. We claim that for some v, β_v initiates at least $(c!k/2)^{1/c}$ physical reads. We prove this by contradiction.

Suppose, for every v, β_v initiates at most p physical reads where $p < (c!k/2)^{1/c}$. Let ρ_v be the sequence of values read, in order, on accessing any given register *for the first time* in β_v. Note that we don't include values obtained from registers which have been read before or been written before in β_v. Clearly, $|\rho_v| \leq p$.

First we assume that the initial configuration is the zero-vector. Therefore, the initial values of all the physical registers are 0. Since α_v contains at most c physical writes, there can be at most c 1's in ρ_v. Clearly, each ρ_v is distinct. Otherwise, if for some v, v' such that $v \neq v'$, $\rho_v = \rho_{v'}$, then a READ in both cases would RETURN the same value, which would be a contradiction. Therefore, $\{\rho_v | v \in V\}$ is a set of k distinct strings of length at most p that differ from the zero-vector in at most c bits. Since $p < (c!k/2)^{1/c}$, this contradicts Lemma 3.8.

In the general case where the initial configuration is *not* the zero-vector, we can no longer claim that ρ_v contains at most c 1's. We therefore define the string δ_v, for every v, as follows. For every bit in ρ_v, if the value is the *same* as the initial value of the register read, place the bit 0 in δ_v. If the value is *different* from the initial value of the register read, place the bit 1 in δ_v. Since α_v contains at most c writes, δ_v can contain at most c 1's. Also, each δ_v is distinct. Now, the same argument as in the previous paragraph holds, with δ_v substituted for ρ_v.

Therefore, for some v, β_v initiates at least $(c!k/2)^{1/c}$ physical reads. This gives our lower bound for R_A. □

Theorem 3.12 presents bounds on the costs of algorithms that are a hybrid of the binary and unary representation algorithms. Using this theorem, we can derive upper bounds on M

and R for c-write algorithms. Theorem 3.13 concerns bounds on W for algorithms that use $\lceil \log k \rceil$ physical registers.

The binary representation algorithm yields an upper bound of $\lceil \log k \rceil$ for R, W, and M. The unary representation algorithm brings down the upper bound for W to 2, while pushing up the bounds for R and M to $\Omega(k)$. This suggests a trade-off between these measures. We can construct a class of algorithms, by borrowing from the two previously mentioned algorithms, that have bounds on R_A and M_A varying from $\Theta(\log k)$ to $\Theta(k)$ and bounds on W_A varying from $\Theta(\log k)$ to $\Theta(1)$.

THEOREM 3.12. *For any m, $1 \leq m \leq k$, there is an algorithm A such that $R_A = \Theta(\log m + k/m)$, $M_A = \Theta(\log m + k/m)$, and $W_A = \lceil \log m \rceil + 2$.*

Proof. We implement our k-ary register by combining an a-ary register and a b-ary register as follows. Let a be the smallest power of 2, which is at least as large as m, i.e., $a = 2^{\lceil \log m \rceil}$. Let $b = \lceil k/a \rceil$. We implement an a-ary register by the *binary* representation method, and a b-ary register by the *unary* representation method. Both these methods have been described earlier. Let the values represented by the a-ary register be in $A = \{1, \ldots, a\}$ and the values represented by the b-ary register be in $B = \{1, \ldots, b\}$. We obtain an ab-ary register by combining these two registers, where the ab values represented are in $A \times B$. Note that $ab \geq k$, so we have our k-ary register.

We consider the bounds of our combination register. The a-ary register uses $\lceil \log m \rceil$ registers and $\lceil \log m \rceil$ physical operations per logical operation. The b-ary register uses $\lceil k/a \rceil$ registers, $\lceil k/a \rceil$ physical reads per logical READ, and 2 physical writes per logical WRITE. Therefore, the total number of registers used is $\lceil \log m \rceil + \lceil k/2^{\lceil \log m \rceil} \rceil$. This is also the number of physical reads per logical READ. There are $\lceil \log m \rceil + 2$ physical writes per logical WRITE. This gives the combined bounds claimed by our theorem. □

The preceding theorem helps us to derive upper bounds for M_S and R_S, where S is the class of c-write algorithms. Choose $m = 2^{c-2}$. Since $c \leq \log k$, it follows that $m \leq k$ and Theorem 3.12 applies. Therefore, there exists an algorithm A such that

(i) $W_A = c$,
(ii) $R_A = c - 2 + \lceil k/2^{c-2} \rceil$, and
(iii) $M_A = c - 2 + \lceil k/2^{c-2} \rceil$.

We thus have the corresponding upper bounds for R_S and M_S, where S is the class of c-write algorithms. Clearly, the upper bounds obtained earlier for the class of 1-write algorithms also hold for c-write algorithms. These new bounds surpass the earlier bounds when $c \geq 3$.

The next theorem states that if an algorithm uses only $\lceil \log k \rceil$ physical registers, then some logical WRITE must use at least $\lceil \log k \rceil$ physical writes.

THEOREM 3.13. *For any algorithm A, if $M_A \leq \lceil \log k \rceil$, then $W_A \geq \lceil \log k \rceil$.*

Proof. Let A be an algorithm with $M_A = \lceil \log k \rceil$. (We have already shown M_A cannot be smaller.) Since the physical registers are binary, $|\mathcal{T}| \leq 2^{\lceil \log k \rceil}$. Recall that for all $v \in V$, there is an $x \in \mathcal{T}$ with $L(x) = v$.

Let U be the subset of $\mathcal{T}$ such that x is in U if and only if there is no $y \neq x$ in $\mathcal{T}$ such that $L(y) = L(x)$. Thus for each configuration x in U, x is the only terminal configuration which has the logical value $L(x)$.

CLAIM 3.14. *There is an $x \in U$ such that $\overline{x} \in \mathcal{T}$. ($\overline{x}$ is the binary string that differs from x in every bit.)*

Proof. Suppose there is no such x. Let $|U| = l$. Each element of U corresponds to a distinct element of V, accounting for l elements of V. The remaining $k - l$ elements of V are represented among the configurations of $\mathcal{T}$ that are not in U and are not the inverse of an element of U. There are at most $2^{\lceil \log k \rceil} - 2l$ of these configurations. There are at least two of these configurations for each remaining element of V. Thus $2^{\lceil \log k \rceil} - 2l \geq 2(k - l)$, which implies $\lceil \log k \rceil \geq \log k + 1$, a contradiction. □

Choose $x \in U$ such that $\overline{x} \in \mathcal{T}$. Let σ be a schedule in $\mathcal{WOC}$ with configuration $\overline{x}$. Suppose $L(x) = v$. Then there is a schedule τ in $\mathcal{WOC}$ of the form σ WRITE(v) α ACK, where α contains no ACK. The configuration of τ must be x since $x \in U$. Thus α contains at least $\lceil \log k \rceil$ writes, and $W_A \geq \lceil \log k \rceil$. □

4. k-ary regular register from binary regular registers. We now shift our attention to regular registers. We would like to implement an n-reader, k-ary, regular register using n-reader, binary, regular registers. Binary regular registers and binary safe registers have the same power. In other words, one can be implemented from the other using one physical register per logical register, at most one physical write per logical WRITE, and one physical read per logical READ [6].

As with safe registers, the problem of implementing k-ary regular registers can also be shown to have a parallel in a combinatorial problem. If there exists an algorithm to implement a k-ary regular register that uses m binary registers, then there is a partial k-coloring of an m-dimensional hypercube with the following restriction. For each colored vertex v, let c be its color. Then, for each color c_i such that $c_i \neq c$ (there are $k - 1$ such colors), there exists a path in the hypercube from v to some vertex v_i with color c_i *all of whose intermediate vertices are colored* c.

This characterization takes care of a slow WRITE that overlaps a number of READs. The path corresponds to the intermediate configurations reached during a WRITE. It makes sure that whatever value is RETURNed by a READ that sees an intermediate configuration preserves the regular property of registers. Note, however, that while this restriction is necessary for an algorithm, it is not sufficient. This is because the restriction doesn't take care of the problem of a slow READ overlapping a number of WRITEs, as we will show later. In particular, our hypercube algorithm for safe registers satisfies this characterization, but cannot be used to implement a regular register. Therefore, this characterization may help us get a lower bound for this problem, but not an upper bound.

Subsection 4.1 shows our independent bounds on R, W, and M. Subsection 4.2 contains our trade-off results. As before, we let $V = \{0, \ldots, k - 1\}$.

4.1. Independent bounds. The following theorem establishes the independent bounds achieved for this problem.

THEOREM 4.1. *The implementation of an n-reader, k-ary, regular register by n-reader, binary, regular registers gives the following independent bounds*:

(i) $R = \lceil \log k \rceil$,

(ii) $1 \leq W \leq \lceil \log k \rceil$, *and*

(iii) $\max\{\lceil \log k \rceil + 1, \lceil 2 \log k - \log \log k \rceil - 2\} \leq M \leq \min\{k - 1, n(3 \log k + 68)\}$.

Proof. The lower bound for R follows directly from a similar proof as the one for safe registers. The lower bound for W is obvious. The lower bound for M is shown in Lemmas 4.4 and 4.5.

The upper bounds on R and W appear simultaneously in the tree algorithm, presented below. However, this algorithm uses $k-1$ physical registers. Lamport [6] describes a complex composition of implementations to achieve an algorithm using $n(3 \log k + 68)$ 1-reader physical registers (recall that n is the number of readers for the logical register). It is unknown whether a better result, for example without the factor of n, is possible by taking advantage of the additional power when the physical registers are n-reader. □

The **modified unary algorithm** is a simple algorithm in [6] that gives upper bounds of $W \leq k$, $R \leq k$ and $M \leq k$. Given registers $X_0, \ldots, X_{k-1}$, the index of the lowest indexed register that has the value 1 determines the k-ary value represented. A READ operation reads $X_0, X_1, \ldots$, in order, until a 1 is returned. It subsequently RETURNs v, where the 1 was read from X_v. A WRITE(v) operation writes 1 in register X_v, and then writes 0 in $X_{v-1}, \ldots, X_0$, in order. (It is possible to optimize the algorithm so as to remove register X_{k-1}.)

Our hypercube algorithm, which we used to implement a k-ary safe register from binary safe registers, cannot be used to implement a k-ary regular register. The reason for this is as follows. In case of a slow READ that overlaps a number of WRITEs, the physical reads initiated by the READ may return a set of register values that do not represent a configuration that occurred during the course of the READ. Thus a logical value may be RETURNed that does not correspond to a value written by an overlapping or last preceding WRITE. A stronger result, stating that no 1-write algorithm using $k - 1$ registers can implement a k-ary regular register from binary regular registers, is proven in Theorem 4.6.

We now present our new **tree algorithm**, which gives the improved bounds of $R \leq \lceil \log k \rceil$, $W \leq \lceil \log k \rceil$, and $M \leq k - 1$. The registers are the nodes in a binary tree. The tree represents a sort of binary search conducted by the READ operation to find the value written. The READ takes a path from the root to a leaf, while the WRITE follows a path starting from a leaf to the root. The path in the tree taken by the READ, along with the values it reads, uniquely defines the value read.

The tree representation of the registers is described as follows. Given any binary tree of k leaves, the internal nodes of the tree correspond to the registers, while the leaves correspond to the k-ary values. Let the leaves of the tree be labeled in some arbitrary manner by the k values in V.

Let v_0 be the initial value of the logical register. The initial values of the physical registers are those that would result from starting with all 0's in the physical registers and then executing a single WRITE(v_0) operation described as follows.

A WRITE(v) operation writes into the set of registers that form the path from the leaf labeled v to the root, beginning with the parent of the leaf, following the path, and ending with the root. The value written to the ith node on the path is 0 (respectively, 1) if the $(i - 1)$-st node on the path is the left (respectively, right) child.

A READ operation reads a set of registers that form a path from the root to a leaf labeled v, for some v, beginning with the root. Suppose the ith node read has value 0 (respectively, 1). If its left (respectively, right) child is a leaf, then v is RETURNed, where v is the label of the leaf. Otherwise, the left (respectively, right) child of the ith node is the $(i + 1)$-st node read.

We just showed that any binary tree with k leaves completely specifies our algorithm. Simple results in graph theory imply that for any k, there exists a binary tree with k leaves, $k - 1$ internal nodes, and height $\lceil \log k \rceil$ (the number of edges in the longest path from the root to a leaf). To obtain the desired complexity bounds, we base our algorithm on one of these trees. Since only internal nodes correspond to registers, $M_A = k - 1$, $R_A \leq \lceil \log k \rceil$, and $W_A \leq \lceil \log k \rceil$.

Figure 3 illustrates a 7-ary register with value 3. The path marked on the tree corresponds to the physical registers read by a logical READ operation.

If k is a power of 2, the registers and values form a *complete* binary tree of height $\log k$. We describe the algorithm, for this special case, formally in Fig. 4. Let $v_m v_{m-1} \ldots v_1$ be the binary representation of the k-ary value v, where $m = \log k$. The root register is labeled with the empty string ϵ. For each register labeled with the binary string l, the strings $l0$ and $l1$ are the labels of its left and right children, respectively. Let the initial value of the logical register be v_0 with its binary representation being $v_{0,m} v_{0,m-1} \ldots v_{0,1}$. Then the initial value of the physical register labeled $v_{0,m} \ldots v_{0,p+1}$ is $v_{0,p}$, for all $p \in \{1, \ldots, m\}$. All other physical registers have initial value 0.

Here, the $\log k$ physical values read by the READ operation form the binary representation of the k-ary number. Clearly, the algorithm has the bounds stated.

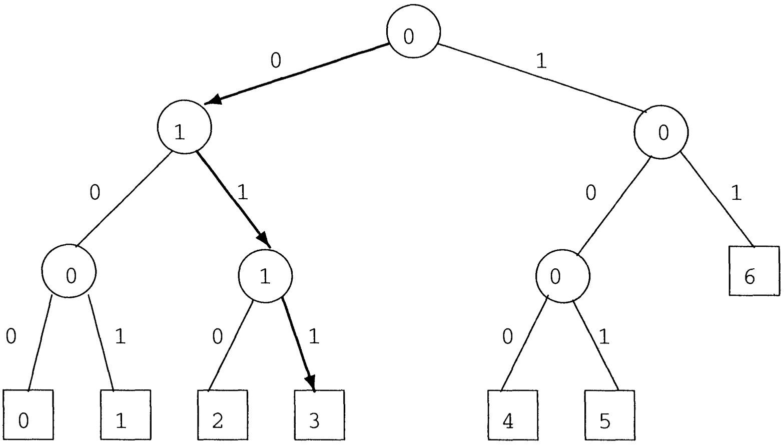


FIG. 3. *An example illustrating the tree algorithm.*

```
To WRITE($v$),
for $p$ := 1 to $m$ do
    write $v_p$ to register $v_m \ldots v_{p+1}$
ACK
```

```
To READ,
for $p$ := $m$ to 1 do
    $v_p$ := read register $v_m \ldots v_{p+1}$
RETURN($v_m \ldots v_1$)
```

FIG. 4. *Tree algorithm for k a power of 2.*

In order to prove the correctness of the tree algorithm, we need some definitions and a lemma. We define a physical read r to **reflect** a physical write w, in a given schedule, if r and w access the same physical register, and either (1) w completely precedes r, or (2) w and r overlap and r returns the value that w writes. We say that a logical READ R **notices** a logical WRITE W if R contains a physical read that reflects a physical write contained in W.

LEMMA 4.2. *Given any schedule of the tree algorithm, and any* READ *R in the schedule, R* RETURNs *the value written by the last* WRITE *W that R notices (note that there is a total order among the* WRITE *operations). If no such* WRITE *exists, R* RETURNs *the initial value.*

Proof. Let R be a READ in some schedule. Suppose R notices no WRITEs. Then every physical read r initiated by R returns the initial value of the physical register read. Therefore, R RETURNs the initial value of the logical register.

Otherwise, R notices some WRITEs. Let W be the last WRITE that R notices. Let s be the last register read by R such that R's read from s reflects W's write to s. Clearly, R reads the value b written by W into s. Otherwise, there is a later WRITE W_1 such that W_1 writes s and R notices W_1; which contradicts the fact that W is the last WRITE that R notices.

Without loss of generality, let $b = 0$. (The argument for $b = 1$ is identical by replacing "left" in the following discussion with "right.")

We claim that s is the last register read by R. Suppose not. Then, R next reads the register t corresponding to the left son of s. Since W wrote b in register s, it must have earlier written to register t. This contradicts the definition of s.

Now, the left son of s must be a leaf node. Let v be the label of this leaf node. Clearly, v is RETURNed by R. Since W writes b into s, the logical value written by W is v. □

THEOREM 4.3. *The tree algorithm implements a k-ary regular register using binary regular registers.*

Proof. We need to argue that our logical k-ary register behaves correctly; i.e., given that our algorithm is implemented using regular binary physical registers, it actually implements a regular k-ary register. Clearly the algorithm has the wait-free property.

Given any schedule, and any READ R in that schedule, we need to prove that R RETURNs the value of one of the WRITE operations it overlaps with or the last preceding WRITE W_1 (or the initial value, in the case that no WRITE completely precedes R). We consider two cases.

Case 1: R notices no WRITEs. Since R reads the root node, and any WRITE must write into the root node, it follows that no WRITE completely precedes R. By Lemma 4.2, R RETURNs the initial value, and this satisfies regularity.

Case 2: R notices some WRITEs. Let W_1 be the last WRITE that R notices. By Lemma 4.2, R RETURNs the value written by W_1. We show that W_1 either overlaps with R or is the last WRITE preceding R. This would satisfy regularity.

Clearly, W_1 cannot completely follow R, since, by the definition of *notice*, W_1 contains a physical write that either precedes or overlaps a physical read contained in R. The only other case to consider is that W_1 precedes another WRITE W_2, which completely precedes R. Since W_1 is the last WRITE that R notices, R does not notice W_2. Since W_2 completely precedes R, R must read the root node after W_2 writes into it, which implies that R does notice W_2. This gives a contradiction. Therefore, W_1 either overlaps with R or is the last WRITE preceding R. □

The tree algorithm simultaneously gives us the best bounds we have for this problem. If the frequencies of READs and WRITEs of all the k values were known in advance, then the number of accesses per READ or WRITE could be optimized by organizing the binary registers as a Huffman tree. For a discussion of Huffman codes, see Hamming [4].

We present our lower bounds for M as follows. Both of the bounds we obtain are significant for different values of k. The bound of Lemma 4.5 supersedes the bound of Lemma 4.4 for $k \geq 55$.

LEMMA 4.4. $M \geq \lceil \log k \rceil + 1$.

Proof. Choose any algorithm A. We assume, for contradiction, that $M_A = \lceil \log k \rceil$. Note that the lower bound for M of $\lceil \log k \rceil$, proved for safe registers, holds here as well. For all $v \in V$, there is a schedule σ_v of A in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Let C_v be the configuration of σ_v; it is easy to see that C_v is stable.

Choose $v \in V$. For each $w \in V$, $w \neq v$, there is a schedule σ_{vw} in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK WRITE(w) β_{vw} ACK, where β_{vw} contains no ACK. Let C_{vw} be the configuration of σ_{vw}; it is easy to see that C_{vw} is stable.

Since only WP takes steps in σ_{vw} and physical writes are done serially, β_{vw} goes through a sequence of stable configurations (corresponding to schedules in $\mathcal{WO}$). By Lemma 2.2, $L_1(C_{vw}) = w$ and $L_1(C_v) = v$. Since $w \neq v$ and L_1 is a function by Lemma 2.1, $C_{vw} \neq C_v$. Thus a stable configuration is reached in β_{vw} that is different than C_v. Let D_{vw} be the first such configuration. D_{vw} and C_v differ in a single bit, i.e., in the value of a single register.

Since there are only $\lceil \log k \rceil$ bits in each configuration, there are only $\lceil \log k \rceil$ configurations that differ in a single bit from C_v. Since there are $k - 1$ values in V different than v, there exist distinct w and u in V such that $D_{vw} = D_{vu}$. Call this configuration D_v. By regularity, $L_1(D_{vw}) \in \{v, w\}$ and $L_1(D_{vu}) \in \{v, u\}$. Thus $L_1(D_v) = v$.

Since $L_1(C_v) = v$, all the C_v's are distinct. Since $L_1(D_v) = v$, all the D_v's are distinct. It is easy to see that $C_v \neq D_w$ for all v and w. Thus there are at least $2k$ distinct stable configurations, requiring at least $\lceil \log k \rceil + 1$ registers. Therefore, we have a contradiction. □

LEMMA 4.5. $M \geq \lceil 2 \log k - \log \log k \rceil - 2$.

Proof. Choose any algorithm A. Let d be the number of registers used in the algorithm.

For all $v \in V$, there is a schedule σ_v of A in $\mathcal{WOC}$ of the form WRITE(v) α_v ACK, where α_v contains no ACK. Let C_v be the configuration of σ_v; it is easy to see that C_v is stable. Clearly, $L_1(C_v) = v$.

We claim that for any two k-ary values v and w, there exists a pair of stable configurations D_v and D_w that differ in exactly one bit such that $L(D_v) = v$ and $L(D_w) = w$. Suppose not. Then, consider the schedule σ_{vw} in $\mathcal{WOC}$ of the form σ_v WRITE(w) β_{vw} ACK, where β_{vw} contains no ACK. Let the configuration of σ_{vw} be D_{vw}. The configuration of σ_v is C_v. Note that D_{vw} is a stable configuration and $L_1(D_{vw}) = w$. Consider the sequence of stable configurations reached by the schedule σ_{vw} starting from C_v and ending at D_{vw}. By the assumption, there exists a stable configuration D_x in the sequence such that $L_1(D_x) = x$ but $x \neq v$ and $x \neq w$. A READ starting at D_x would therefore RETURN x, which violates regularity. This gives a contradiction.

Recall that $\mathcal{S}$ is the set of all configurations resulting from schedules in $\mathcal{WO}$ (only WP takes steps and no physical write is pending). Let c_v be the number of stable configurations C in $\mathcal{S}$ such that $L_1(C) = v$, for each k-ary value v. Let $c = \min\{c_x | x \in V\}$, and let $v \in V$ be such that $c = c_v$. For each value w such that $w \neq v$, there are stable configurations D_v and D_w in $\mathcal{S}$ that differ in exactly one bit such that $L_1(D_v) = v$ and $L_1(D_w) = w$. Since each stable configuration C, such that $L_1(C) = v$, has d neighbors, and there are $(k - 1)$ values w, it follows that $cd \geq k - 1$. Since there are k different values and at most 2^d possible stable configurations, $ck \leq 2^d$. Solving the two inequalities, we obtain that $k^2 - k \leq d\,2^d$, which implies that $2(\log k) \leq d + \log d + 1$, for $k \geq 2$. The last inequality implies that $d \geq 2(\log k) - \log\log k - 2$. □

4.2. Trade-offs. We have the following lower bounds for R and M relating to one-write algorithms. In particular, we show that any one-write algorithm for this problem would require at least k registers. In other words, our hypercube algorithm for safe registers does not work for regular registers.

THEOREM 4.6. *For all algorithms A, if $W_A = 1$ then $R_A \geq k - 1$ and $M_A \geq k$.*

Proof. The lower bound for R_A follows from a similar proof as the one for safe registers. By using a similar argument, we can actually make the additional claim that every READ reads at least $k - 1$ distinct physical registers. We use this claim in the following proof of the bound for M_A.

To show $M_A \geq k$, suppose in contradiction that a one-write algorithm A exists that uses $k - 1$ registers. Then Lemma 3.6 carries over from the safe case, implying that the function L has the rainbow-coloring property. Let C_0 be the initial configuration; clearly, $L(C_0) = v_0$. Consider the following schedule α: READ(1) δ RETURN(1, v_0), where δ consists only of physical actions taken by RP$_1$. We claim that δ does not contain any physical write.

CLAIM 4.7. *The sequence of actions δ does not contain a physical write.*

Proof. Suppose δ does contain a physical write, i.e., $\delta = \delta_1$ write$_i(b)$ δ_2, where δ_1 contains no physical write. Then, there is a schedule of the form

READ(1) δ_1 write$_i(b)$ δ_2 RETURN(1, v_0) READ(1) δ' RETURN(1, v_0),

where δ' contains only physical actions. Let C_1 be the configuration that differs from C_0 only in position i. Then $L(C_1) = v$, for some $v \neq v_0$.

Consider the schedule WRITE(v) γ ACK, where γ contains only physical actions of WP. Then γ consists of a single physical write, to register i (as well as possibly some physical reads). An easy induction shows that

READ(1) δ_1 WRITE(v) γ ACK write$_i(b)$ δ_2 RETURN(1, v_0) READ(1) δ' RETURN(1, v_0)

is a schedule, since there is no physical write in δ_1 and the physical write within the logical

WRITE is "obliterated" by $\text{write}_i(b)$. This violates regularity because the second READ should RETURN v, not v_0. □

Now, we continue with the proof of the theorem. Pick two distinct registers (call them registers i and j) that are read in schedule α.

We define C_1 to be the stable configuration that differs from C_0 in position j, C_3 to be the stable configuration that differs from C_0 in position i, and C_2 to be the stable configuration that differs from C_0 in positions i and j. For all $l \in \{1, 2, 3\}$, C_l is a terminal configuration. Let $L(C_l) = v_l$. It is easy to verify that v_0, v_1, v_2, and v_3 are distinct values in V. Suppose, without loss of generality, that the initial value of both registers i and j is 0. Figure 5 illustrates the relation between the four configurations defined. Adjacent configurations differ in a single bit. The label on the edge between two configurations corresponds to the particular bit in which they differ.

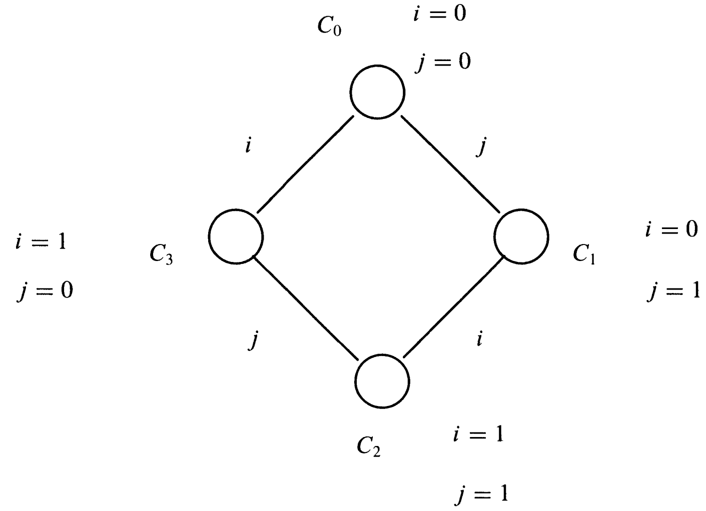


FIG. 5. *Relationship between the four configurations.*

Now, consider the sequences of actions, specified in Table 3, which can be applied at a configuration C_{start} and results in the configuration C_{finish}.

TABLE 3
Sequences for proof of Theorem 4.6.

C_{start}	sequence β	C_{finish}
C_0	$\beta_{01} = \text{WRITE}(v_1)\ \gamma_{01}\ \text{write}_j(1)\gamma'_{01}$ ACK	C_1
C_1	$\beta_{12} = \text{WRITE}(v_2)\ \gamma_{12}\ \text{write}_i(1)\gamma'_{12}$ ACK	C_2
C_2	$\beta_{23} = \text{WRITE}(v_3)\ \gamma_{23}\ \text{write}_j(0)\gamma'_{23}$ ACK	C_3
C_3	$\beta_{32} = \text{WRITE}(v_2)\ \gamma_{32}\ \text{write}_j(1)\gamma'_{32}$ ACK	C_2
C_2	$\beta_{21} = \text{WRITE}(v_1)\ \gamma_{21}\ \text{write}_i(0)\gamma'_{21}$ ACK	C_1

We claim that if we have a schedule σ with the configuration C_{start} and no pending WRITE, we can concatenate an appropriate sequence of actions β (from Table 3) to σ to obtain the schedule σ' with the configuration C_{finish}. The sequence β is a single logical WRITE which consists of a single physical write (and possibly some physical reads)—thus none of the γ_{ab}'s contain any physical writes. It is easy to see that each β exists.

We create a new sequence α' by taking α and inserting certain sequences at certain points, according to the following rules. First, we insert β_{01} immediately before READ(1), resulting in configuration C_1. Then, immediately before each read$_j$ of RP$_1$, if the configuration is C_1, we insert $\beta_{12}\beta_{23}$, resulting in configuration C_3. Immediately before each read$_i$ of RP$_1$, if the configuration is C_3, we insert $\beta_{32}\beta_{21}$, resulting in configuration C_1.

To see that α' is a schedule, it is sufficient to observe that the only time the configuration changes within the schedule is when a sequence β_{ab} is inserted. This follows from the fact, proven in Claim 4.7, that α contains no physical writes. In particular, inserting β_{01} changes the configuration to C_1, inserting $\beta_{12}\beta_{23}$ changes the configuration to C_3, and inserting $\beta_{32}\beta_{21}$ changes the configuration to C_1. We can prove, by a simple induction, that the configuration reached by any prefix of schedule α' up to a read$_i$ by RP$_1$ is always C_1. Similarly, the configuration reached by any prefix of schedule α' up to a read$_j$ by RP$_1$ is always C_3. Therefore, read$_i$ and read$_j$ always return the value 0. It follows that v_0 is the value RETURNed by the READ(1) in the schedule α'. Since, to satisfy regularity, the READ should RETURN v_1, v_2, or v_3, we have a contradiction. □

We conclude this section with a trade-off result relating to a constant number of writes. This follows from the identical result derived in the safe case.

THEOREM 4.8. *For all algorithms A, if $W_A = c$, where $c \leq (\log k)/3$, then $M_A \geq (c!k/2)^{1/c}$ and $R_A \geq (c!k/2)^{1/c}$.*

5. Conclusion. We have demonstrated upper and lower bounds on the number of physical registers, the number of physical reads in a logical read, and the number of physical writes in a logical write, for a variety of multivalued register implementations. In many cases, our bounds are tight. Some of our upper bounds follow from two new algorithms that we present, one for implementing a k-ary safe register out of binary safe registers, and another for implementing a k-ary regular register out of binary regular registers. We also presented several interesting trade-offs between these cost measures, for implementing k-ary registers out of binary registers. The bounds on the number of physical operations can be converted into bounds on the time to perform the logical operations, in terms of the time for the physical operations.

Future work includes finding such bounds for more algorithms, in particular, those involving atomic registers and multi-writer registers. The bounds in this paper on W and M for implementing a k-ary regular register out of binary regular registers are not tight. (Current work shows that the tight bound for W is 1, i.e., that there exists a 1-write algorithm for a k-ary regular register [3].) A final question is what difference does it make, if any, if clocks are available to the read and write processes?

Acknowledgments. We would like to thank Brian Coan for the proof of Lemma 3.7. Akhilesh Tyagi suggested the combinatorial techniques required in proving Theorems 3.10 and 3.11. We are grateful to Hagit Attiya, Paul Beame, Richard Ladner, Mike Saks, Martin Tompa, and George Welch for many helpful conversations that improved some of the results and presentation. The notes [8] from the Distributed Algorithms class taught by Nancy Lynch at MIT served as a useful survey and tutorial of earlier work. The anonymous referees made helpful comments that improved the presentation.

REFERENCES

[1] B. BLOOM, *Constructing two-writer atomic registers*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 249–259.

[2] J. E. Burns and G. L. Peterson, *Constructing multi-reader atomic values from non-atomic values*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 222–231.

[3] S. Chaudhuri, M. Kosa, and J. L. Welch, *Upper and lower bounds for one-write multivalued regular registers*, in Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, December 1991, pp. 134–141.

[4] R. W. Hamming, *Coding and Information Theory*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1986.

[5] G. Kant and J. van Leeuwen, *The file distribution problem for processor networks*, in Lecture Notes in Computer Science 447: Proceedings of the Second Scandinavian Workshop on Algorithm Theory, Springer-Verlag, Berlin, 1990, pp. 48–59.

[6] L. Lamport, *On interprocess communication*, Distributed Computing, 1 (1986), pp. 86–101.

[7] M. Li, J. Tromp, and P. M. B. Vitanyi, *How to Share Concurrent Wait-Free Variables*, manuscript.

[8] N. A. Lynch and K. J. Goldman, *Distributed Algorithms: Lecture Notes for* 6.852, Research Seminar Series MIT/LCS/RSS 5, Massachusetts Institute of Technology, Cambridge, MA, May 1989.

[9] N. A. Lynch and M. R. Tuttle, *Hierarchical correctness proofs for distributed algorithms*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 137–151.

[10] R. Newman-Wolfe, *A protocol for wait-free, atomic, multi-reader shared variables*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 232–248.

[11] G. Peterson, *Concurrent reading while writing*, Association for Computing Machinery Trans. Programming Languages and Systems, 5 (1983), pp. 46–55.

[12] A. K. Singh, J. H. Anderson, and M. G. Gouda, *The elusive atomic register revisited*, in Proceedings of the Sixth Annual Association for Computing Machinery SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Vancouver, Canada, August 1987, pp. 206–221.

[13] J. T. Tromp, *How to Construct an Atomic Variable*, Tech. report CS-R8939, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, October 1989.

[14] A. Tyagi, *The Role of Energy in VLSI Computations*, Ph.D. thesis, Department of Computer Science, University of Washington, Seattle, WA, 1988. Available as UWCS Tech. report 88-06-05.

[15] K. Vidyasankar, *Converting Lamport's regular register to atomic register*, Inform. Process. Lett., 28 (1988), pp. 287–290.

[16] ———, *Concurrent reading while writing revisited*, Distributed Computing, 4 (1990), pp. 81–85.

[17] P. M. B. Vitanyi and B. Awerbuch, *Atomic shared register access by asynchronous hardware*, in Proceedings of the Twenty-Seventh Annual IEEE Symposium on Foundations of Computer Science, Toronto, Canada, October 1986, pp. 233–243.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 355–372, April 1994

PARALLEL INFORMATION DISSEMINATION BY PACKETS*

A. BAGCHI[†][‡], E. F. SCHMEICHEL[§], AND S. L. HAKIMI[†][¶]

Abstract. Each vertex of an undirected graph possesses a piece of information that must be sent to every other vertex. They communicate by sending bounded size packets of messages from one vertex to another. The authors describe parallel algorithms, which accomplish the desired tasks for six prominent architectures. The algorithms are optimal, or nearly so, in every case.

Key words. parallel gossiping, interprocessor communication, algorithms for parallel communication on networks

AMS subject classification. F.2.2

1. Introduction. One of the most important issues in parallel and distributed computing is the issue of communication among the processors. In this paper we will consider a communication problem in networks (or graphs) that is usually termed "gossiping" and that abstracts a large class of communication problems in distributed systems. In the standard gossiping problem, each vertex of a connected graph G begins with a unique piece of information (its message), and the goal is to disseminate this message to every other vertex in the graph. (The precise model of communication for accomplishing this will be discussed in the next paragraph.) Gossiping provides a suitable framework in which to study a number of important issues related to communication in multiprocessor systems. Clearly, gossiping problems arise naturally in studying such matters as load balancing, congestion control, synchronization, and billing. More broadly, the gossiping problem is an abstraction of communication that arises in a large class of distributed computation problems. Thus, optimal solutions to gossiping provide a lower bound for the communication complexity of such problems. Often, the solution to a problem in distributed systems is essentially a solution of gossiping using a different communication model [3], [14]. In addition, the efficiency of an optimal time algorithm for gossiping on a particular topology is a useful measure of that graph's parallel communication capability when viewed as an interconnection network.

Because of its wide applicability and interesting theoretical aspects, gossiping problems have been studied by many researchers. An excellent survey of previous work on gossiping and related problems is provided by Hedetniemi, Hedetniemi, and Liestman [9].

There are two prominent models of communication used in the study of gossiping problems: the telephone model and the mail (or telegraph) model. We may think of each edge (x, y) in a connected graph G as providing a two-way communication link between the end vertices x and y. In the telephone model, communication between x and y can take place in both directions simultaneously; that is, in a single call, x and y may exchange all the messages they possess. By contrast, in the mail model, communication between x and y takes place in only one direction at a time. Thus, in a single transmission x may send y a letter containing all messages known to x, or vice versa, but both letters cannot be sent during the same transmission. In either model, when x and y are communicating with each other, it is assumed that neither is involved in any other activity in the network. In this paper, we will exclusively use the mail model of communication.

*Received by the editors December 10, 1990; accepted for publication (in revised form) November 30, 1992.

[†] Supported by National Science Foundation grant NCR-8716876.

[‡] Bell Communications Research, Red Bank, New Jersey 07701.

[§] Department of Mathematics and Computer Science, San Jose State University, San Jose, California 95192. Supported by National Science Foundation grant DMS-8904520.

[¶] Department of Electrical Engineering and Computer Science, University of California, Davis, California 95616.

In the earliest work on gossiping, the objective was to accomplish gossiping using the minimum number of telephone calls or letters. Let G denote any connected graph with $n \geq 4$ vertices. Bumby [5] and Kleitman and Shearer [10] showed that $2n-4$ (respectively, $2n-3$) telephone calls are both necessary and sufficient to complete gossiping in G if G contains (respectively, does not contain) a cycle of length 4. Conversely, Harary and Schwenk [8] showed that in the mail model of communication, $2n-2$ letters are both necessary and sufficient to complete gossiping in G. A generalization of these results in which the edges are assigned weights has been considered by Wolfson and Segall [14].

The minimum number of telephone calls or letters needed to gossip in a graph G is, however, an inherently sequential measure of the difficulty of gossiping in G. In the sequel, we will instead measure optimality in terms of the overall finishing time, allowing many of the transmissions to occur in parallel. More precisely, we will assume that any set of transmissions (x_1, y_1), (x_2, y_2), ... that form an independent set of edges (i.e., a matching) in G could be performed simultaneously. We will call such a set of simultaneously performed transmissions a *round*, and our goal will be to accomplish gossiping in G in as few rounds as possible, regardless of the number of transmissions actually used.

All previous work on parallel gossiping has assumed transmissions of *unbounded* length; i.e., a vertex is allowed to send everything it knows to an adjacent vertex in a single transmission. The earliest important work considered parallel gossiping in complete graphs using the mail model of communication. Entringer and Slater [6] gave an algorithm to gossip in K_n with unbounded length letters in $\log_\phi n + 5 \doteq 1.44 \log_2 n + 5$ rounds, where $\phi = (1+\sqrt{5})/2$. Later, Even and Monien [7] showed that this number of rounds was optimal to within a small additive constant. In [1], the authors studied parallel gossiping with unbounded length letters in other prominent families of graphs (e.g., trees, hypercubes, and cycles) and presented nearly optimal algorithms for each of the architectures considered. Several of the results in [1] were subsequently improved by Krumme and by Krumme, Cybenko, and Venkataraman [11], [12].

Our goal in this paper is to study parallel gossiping in the mail model of communication with transmissions of *bounded* length. Thus, we will think of each transmission as a letter (or packet) of fixed size. For uniformity, we will assume that the message of each vertex has size 1. As will be seen, the constraint of fixed packet size renders the parallel gossiping problem substantially more challenging. However, the number of rounds required to gossip with fixed size packets in a network is a more realistic measure and gives a good estimate of the real time needed to disseminate information in the network.

It is known [1] that determining the optimal number of rounds for gossiping with unbounded size packets in general graphs is NP-hard; almost certainly, the problem remains NP-hard with packets of fixed size, although this problem remains open. This suggests two directions in which we might profitably proceed. One is the development of efficient approximation algorithms for parallel gossiping with fixed size packets. The other is to consider parallel gossiping with fixed size packets in specific families of graphs (architectures). We have chosen the latter direction in this paper. In particular, we will consider parallel gossiping with fixed size packets in complete graphs, hypercubes, paths, grids, cycles, and binary trees—the latter architecture being by far the most difficult. The algorithms we give will be optimal in the number of rounds, or nearly so, in every case.

While the focus of our paper is on parallel algorithms that assume an observer with global knowledge of the graph, we wish to point out that all of our algorithms can nonetheless be readily implemented as distributed algorithms. In particular, the algorithms for hypercubes and binary trees could be directly incorporated for use in a distributed system. Further consideration of this possibility is left to the reader.

2. Parallel gossiping algorithms using bounded size packets. Let $g(p, G)$ denote the number of rounds needed to gossip in $G = (V, E)$ using packets of size p, assuming all initial messages have size 1. Before considering the various architectures, we state an easy lower bound on $g(p, G)$ that is nonetheless quite useful.

THEOREM 1.

$$g(p, G) \geq \frac{|V|(|V| - 1)}{p\lfloor \frac{V}{2} \rfloor},$$

and the inequality is strict for $p > 1$.

Proof. Each vertex must receive $|V| - 1$ messages, and so $|V|(|V| - 1)$ messages must be received altogether by the vertices. But in a single round, at most $p\lfloor \frac{|V|}{2} \rfloor$ messages altogether can be received. Thus we obtain $g(p, G) \geq |V|(|V| - 1)/p\lfloor \frac{|V|}{2} \rfloor$.

We note that the inequality in Theorem 1 is strict if $p > 1$, since no vertex can send p messages in the first round. □

We now consider how to gossip with fixed size packets in various prominent architectures.

Complete graphs. We begin with the case $p = 1$, which will also prove useful for the general case. Let $\chi'(G)$ denote the edge chromatic number of G.

LEMMA 1.

$$g(1, K_n) = \begin{cases} 2(n-1), & n \text{ even}, \\ 2n, & n \text{ odd}. \end{cases}$$

Proof. Edge color K_n with $\chi'(K_n)$ colors. Then we use the following simple procedure:

```
for $j = 1$ to $\chi'(K_n)$ do
  for each edge $(v, w)$ colored $j$ do in parallel
    begin
      $v$ sends his initial message to $w$ during round $2j - 1$;
      $w$ sends his initial message to $v$ during round $2j$;
    end.
```

This completes gossiping in $2\chi'(K_n)$ rounds. The fact that

$$\chi'(K_n) = \begin{cases} n-1, & n \text{ even}, \\ n, & n \text{ odd} \end{cases}$$

is well known [4].

On the other hand, by Theorem 1 we have

$$g(1, K_n) \geq \frac{n(n-1)}{\lfloor n/2 \rfloor} = \begin{cases} 2(n-1), & n \text{ even}, \\ 2n, & n \text{ odd}. \end{cases} \qquad \square$$

We now turn to the case $p > 1$. It will be convenient to partition the vertices of K_n into $\lceil n/p \rceil$ "row sets" $R_1, R_2, \ldots, R_{\lceil n/p \rceil}$, where we assume $|R_i| = p$, for $1 \leq i \leq \lceil n/p \rceil - 2$ and $|R_i| \geq \frac{p}{2}$, for $\lceil n/p \rceil - 1 \leq i \leq \lceil n/p \rceil$. If the vertices in R_i are say $\{v_{i,1}, v_{i,2}, \ldots\}$, we define the p "column sets" $C_1, \ldots, C_p$ by $C_j = \{v_{1,j}, v_{2,j}, \ldots\}$. Note that $C_j = \lceil n/p \rceil$, for $1 \leq j \leq \frac{p}{2}$; see Fig 1. Our gossiping algorithm is now easily described.

		C_1 ↓	C_2 ↓	C_3 ↓	C_4 ↓	C_5 ↓
R_1	→	$v_{1,1}$	$v_{1,2}$	$v_{1,3}$	$v_{1,4}$	$v_{1,5}$
R_2	→	$v_{2,1}$	$v_{2,2}$	$v_{2,3}$	$v_{2,4}$	$v_{2,5}$
R_3	→	$v_{3,1}$	$v_{3,2}$	$v_{3,3}$	$v_{3,4}$	
R_4	→	$v_{4,1}$	$v_{4,2}$	$v_{4,3}$		

FIG. 1. *Partitioning K_{17} for $p = 5$.*

```
algorithm {gossiping in $K_n$ with $p$ packets}
begin
(1) for $i = 1$ to $\lceil n/p \rceil$ do in parallel gossip in $R_i$;
(2) for $j = 1$ to $p$ do in parallel gossip in $C_j$;
    {if $|C_j| = \lceil n/p \rceil$, then each node in $C_j$ knows the cumulative message after (2)}
(3) for $i = 1$ to $\lceil n/p \rceil$ do in parallel
        for $j = 1$ to $\lfloor p/2 \rfloor$ do in parallel
           begin
           (i) $v_{i,j}$ sends $v_{i,j+\lceil p/2 \rceil}$ (if the latter exists) a packet containing the messages
               in $R_{\lceil n/p \rceil - 1}$;
           (ii) $v_{i,j}$ sends $v_{i,j+\lceil p/2 \rceil}$ (if the latter exists) a packet containing the messages
               in $R_{\lceil n/p \rceil}$
           end
end. {gossiping in $K_n$ with $p$ packets}
```

It is easy to verify that the algorithm completes gossiping in a correct way.

For the analysis, note that since $|R_i| \leq p$ for all i, we can use the algorithm of Entringer and Slater [6] to complete Step (1) in $\log_\phi p + 5 = 1.44 \log_2 p + 5$ rounds. Each node begins Step (2) knowing $\leq p$ messages, which all can fit in a single packet. So for gossiping in C_j, we can apply the result in Lemma 1 when $p = 1$ in an obvious way. Thus Step (2) takes at most $2|C_j| \leq 2\lceil n/p \rceil \leq 2\left(\frac{n}{p} + 1\right)$ rounds. Finally Step (3), if needed, takes two rounds. In summary, we obtain the following theorem.

THEOREM 2. $g(p, K_n) \leq \frac{2n}{p} + 1.44 \log_2 p + O(1)$.

We now show that the above algorithm is optimal to within a small constant number of rounds.

THEOREM 3. $g(p, K_n) > \frac{2n}{p} + 1.44 \log_2 p - 3$.

Proof. At any time during the running of an algorithm, let the *knowledge vector* be the n-vector whose ith component is the number of messages known to the ith vertex at that time. Let $\overline{v}_t$ denote the knowledge vector after t rounds, so that initially the knowledge vector is $\overline{v}_0 = \langle 1, 1, \ldots, 1 \rangle$. At the conclusion, the knowledge vector will be $\langle n, n, \ldots, n \rangle$.

Consider the earliest round r at which the "total knowledge" (i.e., the L_1-norm of the knowledge vector) is at least pn. Since $\|v_r\|_1 \geq pn$, it follows easily that the euclidean norm $\|\overline{v}_r\|_2 \geq \|\langle p, p, \ldots, p \rangle\|_2 = p\sqrt{n}$. Since $\|\overline{v}_0\|_2 = \sqrt{n}$, we see that the euclidean norm of the knowledge vector increases by a factor of at least p during the first r rounds.

LEMMA. (Even and Monien [7]). *In a single round, the euclidean norm of the knowledge vector increases by a factor of at most ϕ.*

Sketch of Proof of the Lemma (when n is even). Let $\overline{v}_t = \langle x_1, x_2, \ldots, x_n \rangle$. By suitably permuting the indices, we may assume that $\overline{v}_{t+1} = \langle x_1+x_2, x_2, x_3+x_4, x_4, \ldots, x_{n-1}+x_n, x_n \rangle$; i.e., $\overline{v}_{t+1} = A\overline{v}_t$, where

$$A = \begin{pmatrix} \begin{matrix} 1 & 1 \\ 0 & 1 \end{matrix} & & 0 \\ & \ddots & \\ 0 & & \begin{matrix} 1 & 1 \\ 0 & 1 \end{matrix} \end{pmatrix}.$$

Thus $\|\overline{v_{t+1}}\|_2 \leq \|A\|_2 \|\overline{v_t}\|_2$, where

$$\|A\|_2 = \sup_{\overline{x} \neq \overline{0}} \frac{\|A\overline{x}\|_2}{\|\overline{x}\|_2}$$

denotes the euclidean norm of A. But it is well known [13] that $\|A\|_2 = \sqrt{|\lambda_{\max}\left(A^TA\right)|}$. Since

$$A^TA = \begin{pmatrix} \begin{matrix} 1 & 1 \\ 1 & 2 \end{matrix} & & 0 \\ & \ddots & \\ 0 & & \begin{matrix} 1 & 1 \\ 1 & 2 \end{matrix} \end{pmatrix}$$

has maximum eigenvalue $\frac{1}{2}(3+\sqrt{5})$, we obtain $\|A\|_2 = \phi$, completing the proof of the lemma when n is even. For n odd, we refer the reader to [7].

It follows immediately from the lemma that $r \geq \lceil \log_\phi p \rceil$. Thus after $\lceil \log_\phi p \rceil - 1$ rounds the total knowledge is less than pn. So after the first $\lceil \log_\phi p \rceil - 1$ rounds, the total knowledge must increase by more than $n^2 - pn$. Since the total knowledge increases by at most $p\lfloor n/2 \rfloor$ per round, it follows that after the first $\lceil \log_\phi p \rceil - 1$ rounds, we need more than $(n^2 - pn)/p\lfloor n/2 \rfloor \geq \frac{2n}{p} - 2$ additional rounds. Thus $g(p, K_n) > \frac{2n}{p} + \log_\phi p - 3 > \frac{2n}{p} + 1.44 \log_2 p - 3$. □

Hypercubes. Let Q_d denote the d-dimensional hypercube with $n = 2^d$ vertices. We think of the vertices of Q_d in the usual way, as d-tuples of 0's and 1's, with two vertices adjacent if they differ in exactly one component. Let us say an edge (v, w) belongs to dimension j if v and w differ in the jth coordinate. We have the following easy algorithm for gossiping in Q_d.

```
algorithm {gossiping in $Q_d$ with $p$ packets}
  begin
    for $j = 1$ to $d$ do
      for all edges $(v, w)$ belonging to dimension $j$ do in parallel
        begin
          $v$ sends $w$ all messages known to $v$ using $\lceil 2^{j-1}/p \rceil$ rounds;
          $w$ sends $v$ all messages known to $w$ but not yet known to $v$
            using $\lceil 2^{j-1}/p \rceil$ rounds;
        end
  end. {gossiping in $Q_d$ with $p$ packets}
```

The correctness of the algorithm is immediate. The algorithm requires

$$2\sum_{j=1}^{d}\lceil 2^{j-1}/p\rceil \le 2\sum_{j=1}^{d}\left(\frac{2^{j-1}}{p}+1\right)=2\left[\frac{n}{p}\left(\sum_{j=1}^{d}\frac{1}{2^{d-j+1}}\right)+d\right]$$
$$=\frac{2n}{p}+2\log_2 n-\frac{2}{p}$$

rounds. If p is a power of 2, say $p = 2^c$, then the number of rounds becomes

$$2\sum_{j=1}^{d}\lceil 2^{j-1}/p\rceil = 2c+2\sum_{j=c+1}^{d} 2^{j-c-1} = 2\left(c+2^{d-c}-1\right)=\frac{2n}{p}+2\log_2 p-2.$$

By way of comparison, we have $g(p, Q_d) \ge g(p, K_n) > \frac{2n}{p} + 1.44\log_2 p - 3$. Thus our algorithm is close to optimal for gossiping in Q_d, especially when p is a power of 2.

We summarize our results.

THEOREM 4. (i) $g(p, Q_d) \le \frac{2n}{p} + 2\log_2 n - \frac{2}{p}$.

(ii) *If p is a power of* 2, *then* $g(p, Q_d) \le \frac{2n}{p} + 2\log_2 p - 2$.

Paths. Let P_n denote the path with n vertices. We begin by showing that for $p = 4$ (and a priori $p > 4$), we can gossip in P_n in $\text{diam}(P_n) + O(1)$ rounds; this is clearly optimal up to an additive constant since $g(p, G) \ge \text{diam}(G)$ for any p and G.

Let $p = 4$, and for simplicity of presentation assume $n \equiv 2 \pmod 8$; the modifications needed for other values of n will be easy.

Partition the vertices of P_n into $\frac{n-2}{4}$ groups of four vertices each, and one group of two vertices (the center two vertices), as shown in Fig. 2. Call the edge immediately to the left of the center edge the *crossover edge*.

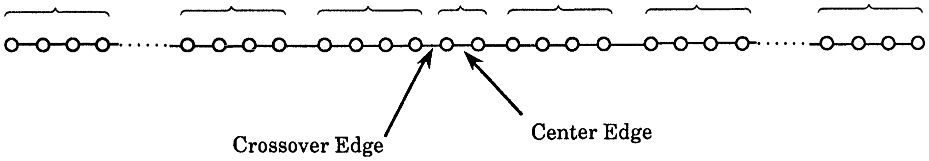


FIG. 2. *Groups for path algorithm, $p \ge 4$.*

Consider any group that occurs to the right of the crossover edge. A packet will begin at the right vertex of the group, and then move steadily to the left (except for "collisions" with other packets, as discussed below). We will call such a packet a *major packet.* It will pick up the messages of the vertices in its own group and proceed to the left endpoint of the path without picking up any additional messages. Similarly, a *minor packet* will begin at the left endpoint of this group and move steadily to the right in a similar way, picking up only the messages in its own group. There are symmetric definitions for a major and minor packet for each group to the left of the crossover edge.

The protocol when collisions occur between packets moving in opposite directions is as follows:

(1) A major packet is never delayed by a minor packet; the minor packet waits two rounds for the major packet to pass.

(2) When two major packets collide, the one that started farther from the crossover edge has priority, and the other waits two rounds until it passes.

It is readily checked that for two packets moving to the right, the one that started farther to the right will always remain ahead, i.e., packets moving to the right will not "pile up." The same holds for any two packets moving left.

Since the major packet from the right end vertex is never delayed, it follows by the above paragraph that all left-moving packets will have reached their destination in $\mathrm{diam}(P_n)$ rounds. The major packet from the left end vertex is delayed only by a collision with the major packet from the right end vertex. Thus all packets moving right will reach their destination in just $\mathrm{diam}(P_n) + 2$ rounds.

Consider now the case when $p = 3$; we assume for simplicity that $n \equiv 2 \pmod 6$. We use the above approach, except that we change the size of the large groups from 4 to 3 and delay the start of the major packets. In particular, if the major packets to the right of the crossover edge are ordered by increasing distance from the crossover edge, then the start of the jth major packet to the right is delayed $j - 1$ rounds. (The intuition is to create the same spread between the left-moving major packets that we had when the large groups had size 4). Similarly, we delay the start of the jth major packet to the left of the crossover edge by j rounds. In particular, the major packets at the end vertices of the path are delayed by $\frac{n}{6} + O(1)$ rounds. The minor packets start as soon as possible, and the protocol for collisions is exactly as above. Thus we have $g(3, P_n) \leq \frac{7}{6}n + O(1)$.

For $p = 1$ and 2, we use a different strategy; for simplicity of presentation we assume n is even. The idea is to take a census at the $(\frac{n}{2} + 1)$th vertex from the left end (denoted v_0 in the sequel) and then to broadcast from v_0 the messages from each half of the path to the other half. At the same time, we disseminate the messages of the vertices in each half to the other vertices in the same half. We illustrate the method for $p = 1, n = 6$ and $p = 2, n = 8$ in Fig. 3, in which the number(s) above an arrow indicates the number of the unit(s) whose messages are being sent.

For $p = 1$, it takes $n - 1$ rounds to complete a census at v_0, another $n + 1$ rounds until v_0 sends out its final packet (v_0 needs to send its own message in both directions), and a final $\frac{n}{2} - 2$ rounds until the last packet sent by v_0 reaches its final destination. Thus $g(1, Pn) \leq \frac{5}{2n} + O(1)$.

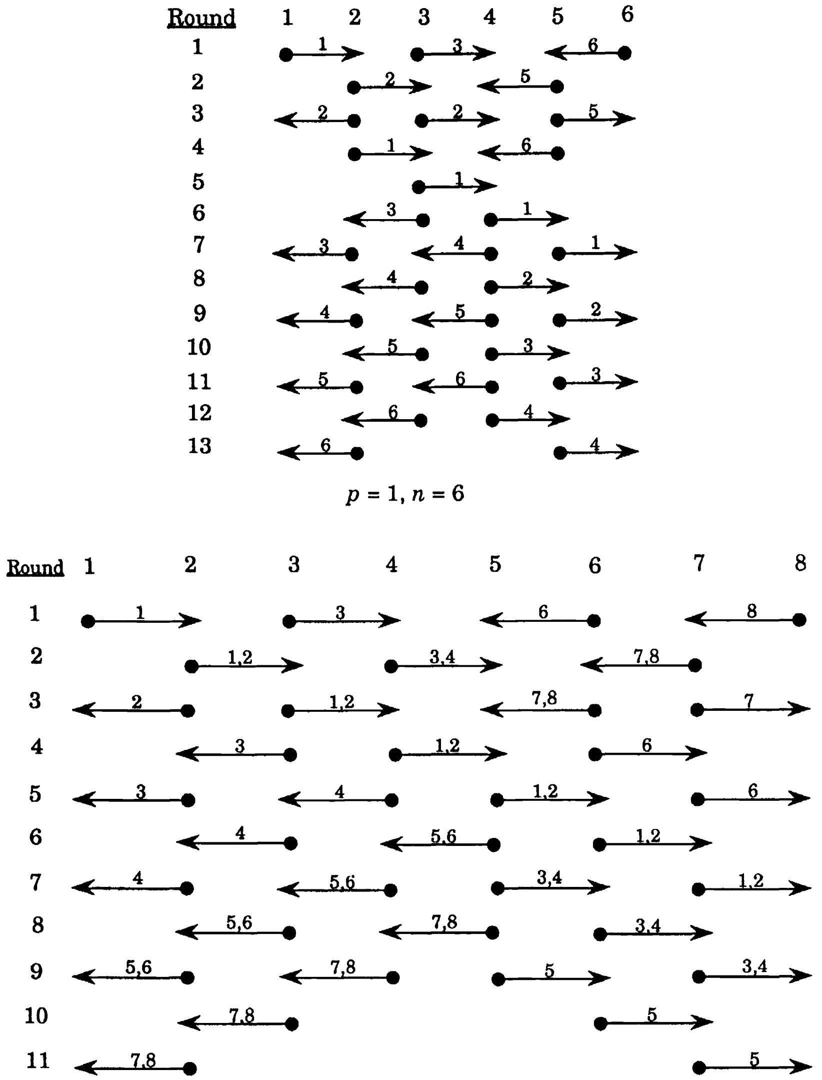
For $p = 2$, it takes $\frac{n}{2}$ rounds to complete a census at v_0, $\frac{n}{2} + 1$ more rounds until v_0 sends out its last packet, and a final $\frac{n}{2} - 2$ rounds until this last packet reaches its final destination. Thus $g(2, P_n) \leq \frac{3}{2n} + O(1)$.

We now show that these upper bounds for $g(p, P_n)$ are optimal to within a small additive constant. For $p \geq 4$, this is immediate. For $p \leq 3$, the optimality follows by the following result.

LEMMA 2.

$$g(p, P_n) \geq \left\lceil \frac{n-1}{p} \right\rceil + \left\lceil \frac{n+1}{p} \right\rceil + \left\lceil \frac{n}{2} - 2 \right\rceil .$$

Proof. Let v_0 be a centermost vertex on P_n. Note that v_0 needs to receive at least $\lceil (n-1)/p \rceil$ crucial packets (i.e., packets bringing v_0 a new message) and to send $\geq \lceil (n+1)/p \rceil$ crucial packets (i.e., packets bringing the recipient a new message). Clearly the last crucial packet sent by v_0 must occur after the last crucial packet received by v_0, and so the last crucial packet sent by v_0 cannot occur before round $\lceil (n-1)/p \rceil + \lceil (n+1)/p \rceil$. But the last crucial packet sent by v_0 contains a message that needs $\geq \frac{n}{2} - 2$ rounds to reach the appropriate end vertex of the path. This proves the lemma. □



$p = 1, n = 6$

$p = 2, n = 8$

FIG. 3.

We summarize our results as follows.

THEOREM 5.

$$g(p, P_n) = \begin{cases} 5/2n + O(1), & p = 1, \\ 3/2n + O(1), & p = 2, \\ 7/6n + O(1), & p = 3, \\ \operatorname{diam}(P_n) + O(1), & p \geq 4. \end{cases}$$

Rectangular grids. Let $G_{m,n}$ denote the grid with m rows and n columns. Since $G_{m,n}$ contains mn vertices, it follows by Theorem 3 that $g(p, G_{m,n}) > \frac{2mn}{p} + 1.44 \log_2 p$. A natural way to gossip in $G_{m,n}$ would be: "Gossip in each of the columns in parallel, and then gossip in each of the rows in parallel." Such an approach uses $\frac{5}{2}\frac{mn}{p}$ rounds. This bound can be asymptotically improved. We present below a slightly more complex algorithm which yields $g(p, G_{m,n}) \leq \frac{2mn}{p} + O(m+n)$.

We begin with the following result.

LEMMA 3. *For n odd*, $g(1, G_{3,n}) \leq 6n+2$.

Proof. Let $n = 2k+1$. Label the units in row r by $\mu_{(r-1)n+1}, \dots, \mu_{rn}$ from left to right, for $r = 1, 2, 3$. We will term a unit μ_i odd (respectively, even) if i is odd (respectively, even). Note that there are $3k+2$ odd units and $3k+1$ even units.

If we delete any odd unit from G, the resulting graph is Hamiltonian. In fact, for each unit μ_i, $i = 1, 3, 5, \dots, 2n+1$, $G - \mu_i$ contains a unique directed Hamiltonian cycle, denoted C_i, which contains the directed path $\mu_{3n}, \mu_{3n-1}, \mu_{3n-2}, \dots, \mu_{2n+1}$. The algorithm will employ only the cycles $C_1, C_3, C_5, \dots, C_{2n-1}$. We note two useful facts about this collection of cycles:

(1) Let μ_i be any even unit. The even unit which immediately follows μ_i on C_j is the same for all $j = 1, 3, 5, \dots, 2n-1$.

(2) Let μ_i be any odd unit. The even units which immediately precede and follow μ_i on $C_j (j \neq i)$ are the same for all $j = 1, 3, 5, \dots, 2n-1$ with $j \neq i$.

The algorithm is conveniently described in two stages. During the first (second) stage, each unit will learn the message of all the even (odd) units.

Stage 1 will begin with the following $6k$ rounds.

```
for $i = 0$ to $k - 1$ do
begin
    during rounds $6i + 1$ and $6i + 3$, every even unit transmits the last message it
        learned to the following (odd) unit on $C_{2i+1}$;
    during rounds $6i + 2$ and $6i + 4$, every odd unit transmits the last message it
        learned to the following (even) unit on $C_{2i+1}$;
    {during these four rounds, we use the cycle $C_{2i+1}$ which omits an odd unit
        in the first row}
    during round $6i + 5$, every even unit transmits the last message it learned to
        the following (odd) unit on $C_{n+2i+2}$;
    during round $6i + 6$, every odd unit transmits the last message it learned to
        the following (even) unit on $C_{n+2i+2}$;
    {during these two rounds, we use the cycle $C_{n+2i+2}$ which omits an odd unit
        in the second row}
end
```

It is easy to see by (1) that after these $6k$ rounds, every even unit knows the messages of all the other even units. Similarly by (2), after $6k$ rounds each odd unit in the first (respectively, second, third) row knows the messages of all the even units with at most two (respectively, one, zero) exceptions. Thus we can easily finish Stage 1 in three additional rounds, for a total of $6k+3$.

In Stage 2, we disseminate the messages of the odd units. In the first round we use C_1, letting each odd unit (except μ_1) send its message to an even unit. We then use a scheme analogous to the one above, with four rounds on C_{n+2}, two rounds on C_3, four rounds on C_{n+4}, two rounds on C_5, etc. The key change is that μ_1, which belongs to all these cycles except the initial C_1, delays sending the message just received from μ_{n+1} for two rounds. In particular, μ_1 sends its own message at round 3. So Stage 2 is analogous to Stage 1 except for this two-round delay. It follows that Stage 2 can be completed in $6k+5$ rounds, and so altogether we can gossip in $G_{3,n}$ in just $12k+8 = 6n+2$ rounds as asserted. □

The algorithm is now quite easy to describe. First gossip in each of the columns in parallel; this takes $\leq \frac{5}{2}m + O(1)$ rounds, by Theorem 5. Next, partition the rows into sets of two consecutive rows {Row 1, Row 2}, {Row 3, Row 4}, ..., with a final set of three rows if n is odd. For each pair {Row $2j-1$, Row $2j$}, note that the vertices in the odd (respectively, even) positions in Row $2j-1$ (respectively, Row $2j$) together know all the messages. These vertices are darkened in Fig. 4. It is now easy to complete gossiping in these two rows in $2n\lceil m/p \rceil \leq 2n\left(\frac{m}{p}+1\right)$ rounds, using the directed cycle in Fig. 4 and moving the m messages which begin at each darkened vertex one position along the cycle every $\lceil m/p \rceil$ rounds.

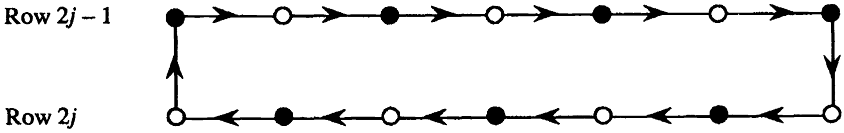


FIG. 4.

For the final set of three rows (if it exists), we assign, to each of the three vertices in a column, $\lceil m/3 \rceil$ messages from that column in the original grid so that together these three vertices are assigned all m original messages from that column. We now simulate the algorithm in Lemma 3 in an obvious way, simulating each transmission in the Lemma by $\lceil m/3p \rceil$ transmissions here. We thus complete gossiping in the final three rows in $(6n+2)\cdot\lceil m/3p \rceil = \frac{2mn}{p} + O(m+n)$ rounds. We have thus proved Theorem 6.

THEOREM 6. $g(p, G_{m,n}) \leq \dfrac{2mn}{p} + O(m+n)$.

Cycles. Let C_n denote the cycle on n vertices. It will be convenient to consider the case when p is even first.

We begin by noting that

$$g(2, C_n) = \begin{cases} n, & n \text{ even}, \\ n+1, & n \text{ odd}; \end{cases}$$

we leave the easy proof of this to the reader, mentioning only that the lower bound follows from the remark following Theorem 1. Suppose therefore that $p \geq 4$. For simplicity of explanation, let us also assume for the moment that p divides n. Label the vertices of C_n in clockwise order as $v_{11}, v_{12}, \ldots, v_{1p}, v_{21}, v_{22}, \ldots, v_{2p}, \ldots, v_{n/p,1}, \ldots, v_{n/p,p}$. We define group j to be the vertices with first index j. We now give a high level description of our gossiping algorithm for C_n under these assumptions. For each j, $1 \leq j \leq n/p$, a packet will begin at $v_{j,1}$ (respectively, $v_{j,p}$) and move clockwise (respectively, counterclockwise), first picking up all the messages of the vertices in group j and then continuing without picking up additional messages to the vertex which is, roughly, diametrically opposite the vertex $v_{j,p/2}$. We illustrate the movement of these two packets in Fig. 5. Whenever two packets moving in opposite directions "collide," we delay each packet one round in allowing the other packet to pass. If a packet moved without collisions, it would take exactly $\frac{n}{2}+\frac{p}{2}-1$ rounds to reach its final destination. But each packet will collide with $\leq \frac{n}{p}$ packets moving in the opposite direction. Thus $g(p, C_n) \leq \frac{n}{2}+\frac{n}{p}+\frac{p}{2}-1$ when p is even and $p|n$.

If p does not divide n, we instead partition C_n into $\lceil n/p \rceil$ groups of consecutive vertices, each group containing at most p, but at least 2, vertices. We then mimic the algorithm above, holding up packets as they are about to exit groups with $\geq p$ vertices so that all packets leave the

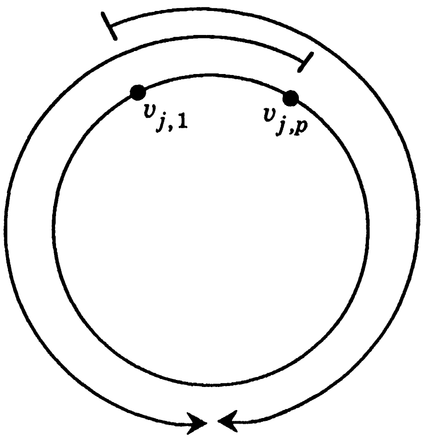


FIG. 5.

groups in synchrony. With this approach, it is easy to see that $g(p, C_n) \leq \lceil n/2 \rceil + \lceil n/p \rceil + \frac{p}{2} - 1$ for any even p.

Consider now the case when p is odd. For $p = 1$, it is easy to show that

$$g(1, C_n) = \begin{cases} 2n - 2, & n \text{ even}, \\ 2n, & n \text{ odd}. \end{cases}$$

For $p \geq 3$, we simply define the packet size to be $p - 1$ (even) and apply the algorithm above. We summarize our result below.

THEOREM 7.

(i) $g(1, C_n) = \begin{cases} 2n - 2, & n \text{ even}, \\ 2n, & n \text{ odd}. \end{cases}$

(ii) $g(p, C_n) \leq \begin{cases} \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{p} \right\rceil + \frac{p}{2} - 1, & \text{if } p \text{ is even}, \\ \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n}{p-1} \right\rceil + \frac{p-1}{2} - 1, & \text{if } p > 1 \text{ is odd}. \end{cases}$

Although the bound above is tight for $p = 1$ and 2, the only lower bound we have when $p \geq 3$ is the trivial $g(p, C_n) \geq 1 + \operatorname{diam}(C_n) = 1 + \lfloor n/2 \rfloor$.

Binary trees. Let B_n denote the complete binary tree on n vertices. We begin by describing an easy algorithm for gossiping in B_n. Color the edges of B_n with three colors, say $\{0, 1, 2\}$. If $r \equiv j (\text{mod } 3)$, the edges colored j will be termed *active* during round r. Throughout the algorithm each node v will maintain a queue $Q(v)$ which initially contains just v's message. As usual, each node also maintains a list of all messages received in its local memory, which is separate from the queue. As messages arrive at v from v's sons, they are inserted in $Q(v)$ in the order received. Let $f(v)$ denote the father of v in B_n. Suppose $(v, f(v))$ is active during round r. If $Q(v)$ is nonempty, then v sends $f(v)$ the first $\min\{p, |Q(v)|\}$ messages in $Q(v)$ in a packet to $f(v)$. Otherwise, if $f(v)$ knows any messages not yet known to v, then $f(v)$ sends v a packet containing as many messages, not yet known to v, as possible. The algorithm halts when each unit knows all the messages. Our goal is to show that this algorithm is optimal to within $O(1)$ rounds if n is sufficiently large relative to p.

We will establish the following upper bound for the running time of the algorithm.

THEOREM 8. *If $p \geq 2$ and $n \geq 4p^{3/2\log_\phi 2}$, the above algorithm uses at most $\frac{3n}{p} + 2\log_2 n + \log_\phi p + O(1)$ rounds.*

Although the algorithm given above is quite easy, the proof of Theorem 8 is rather involved. Before formally establishing the complexity of the algorithm, we will informally discuss the execution of the algorithm and the ideas behind the analysis.

Let r_1 and r_2 denote the two sons of the root $f(r_1)$. Suppose the edge $(r_1, f(r_1))$ is colored 1. It is easily checked that in rounds $1, 4, 7, \ldots, r_1$ sends to $f(r_1)$ packets containing $1, \min\{p, 6\}, \min\{p, 26\}, \ldots,$ messages, respectively. There is a similar pattern for the number of messages sent by any vertex u to $f(u)$ in successive early transmissions. We will establish that if n is large enough in terms of p, then r_1 sends a packet with $\min\{p, 2F_{3k+1}\}$ messages to $f(r_1)$ during round $3k + 1$, for $k = 1, 2, \ldots,$ until r_1 sends $f(r_1)$ at most one final packet with $< p$ messages, where F_j is the jth Fibonacci number. Once r_1 has sent $f(r_1)$ all the messages in the subtree T_{r_1} of T rooted at r_1, then $f(r_1)$ starts sending r_1 the messages from $T_{r_2} \cup \{f(r_1)\}$. Using the above facts, it is easy to show that the root and both of its sons will have learned the cumulative message by round $R + O(1)$, where $R = \frac{3n}{p} + \log_\phi p$.

Now consider vertex x in Fig. 6. Vertex x must receive from x_1 and x_2 all the messages in subtrees T_1 and T_2, respectively. Once x knows all the messages in T_1 (respectively, T_2), x begins sending to x_1 (respectively, x_2) the messages from subtree T_2 (respectively, T_1). Meanwhile, x continues to send messages from $T' = T_1 \cup T_2 \cup \{x\}$ to $f(x) = x_3$. When x_3 knows all the messages from T', it starts sending to x the messages from $T - T'$. Once x starts receiving these messages, it sends them down to x_1 and x_2. Note that if x has finished sending all the messages from T_1 (respectively, T_2) to x_2 (respectively, x_1) before receiving any message from x_3, then the messages from x_3 will encounter no delay at x. If each downward moving packet encountered no delay at any vertex in the tree, then of course gossiping would be completed in $R + 2\log_2 n + O(1)$ rounds. It will be shown, however, that the total delay encountered by any packet moving down the tree is just $O(1)$ rounds, and thus the desired bound of $R + 2\log_2 n + O(1)$ rounds will be established.

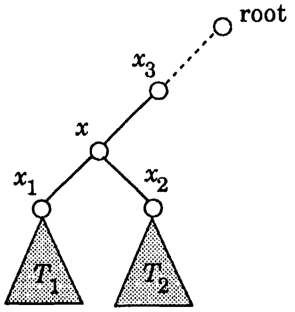


FIG. 6.

To establish a delay of $O(1)$ rounds on any downward moving packet, we will first study the number of messages that are sent by a vertex v to $f(v)$ in successive transmissions (the so-called "upsequence" of the edge $(v, f(v))$). It will be shown that there are only a few types of upsequences. One possible upsequence occurs if v sends $f(v)$ first some packets containing less than p messages, then a series of full packets (containing p messages), and finally at most one packet with less than p messages. We mentioned above that the two edges incident to the root will have this type of upsequence if n is sufficiently large. Indeed, this will be the upsequence of any edge $(v, f(v))$ that is "high enough" in the tree (i.e., the distance h from v

to its leaf descendants is large enough). On the other hand, if this is not the upsequence type of $(v, f(v))$, then we will show that v will have completed sending all messages in T_v to $f(v)$ by round $2h + 3$. Once $f(v)$ knows the messages in T_v, $f(v)$ begins sending the messages from $T - T_v$ to v in packets containing p messages, and v will know the cumulative message by round $R + 2\log_2(n - h) + O(1)$.

Before giving the actual proof, it will be useful to develop a number of lemmas. On any edge e of B_n, packets are first sent upward (towards the root) and then downward later on. The first lemma concerns the number of messages which are sent upward on e in successive packets.

LEMMA 4. *Suppose a packet is sent upward on e during round r containing $up_e(r)$ messages. If $up_e(r) > up_e(r+3)$, then $up_e(r+3) > up_e(r+6) > \cdots$, as long as packets are being sent upward on e.*

Proof. For any edge e, let $r(e)$ denote the first round r in which $up_e(r) < up_e(r-3)$. It is easy to see that if e_1, e_2 are two edges colored the same with e_1 higher in the tree than e_2, then $r(e_1) \geq r(e_2)$.

We now proceed by induction on the height of the edge e above the leaves. Suppose the lemma holds for all edges of height at most h, and let $e = e_0$ have height $h + 1$, with say $r(e_0) = r + 3$. We need to show that $up_e(r+3) > up_e(r+6)$.

We see clearly in Fig. 7 that $r(e_1) \leq r + 2$ or $r(e_2) \leq r + 2$ or both. Without loss of generality suppose $r(e_1) \leq r+2$ and color$(e_1) \equiv r + 1 \pmod 3$. By the induction hypothesis

$$(1) \qquad up_{e_1}(r+1) > up_{e_1}(r+4).$$

Suppose (again, see Fig. 7) that color$(e_0) =$ color(e_3) and color$(e_1) =$ color(e_4). By the observation in the preceding paragraph $r(e_3) \leq r(e_0)$ and $r(e_4) \leq r(e_1)$. Thus

$$(2) \qquad up_{e_3}(r) > up_{e_3}(r+3),$$

$$(3) \qquad up_{e_4}(r+1) > up_{e_4}(r+4).$$

From (2) and (3) we get

$$(4) \qquad up_{e_2}(r+2) > up_{e_2}(r+5),$$

and from (1) and (4) we get the desired result. □

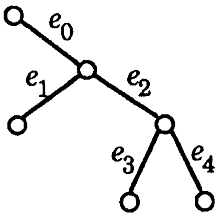


FIG. 7.

Given an edge e colored 0, we call the sequence $(up_e(3), up_e(6), \ldots)$ the *upsequence* of e. Analogous definitions give the upsequence of edges colored 1 or 2. By Lemma 4, the upsequence of each edge is one of the following three types. (We use formal language notation to describe the three types, letting ϵ denote the empty string, and $(<p)$ denote the set of positive integers less than p.)

(1) All-partial: $(<p)^*(<p)$,
(2) Good: $(<p)^*p^*p\{<p, \epsilon\}$,
(3) Bad: $(<p)^*p^*p(<p)^*(<p)^2$.
We will call an edge all-partial, good, or bad according to the type of its upsequence.

Our immediate goal is to show that we can bound the height (above the leaves) of bad edges in B_n in terms of p. As we shall see, if $n \geq 4p^{3/2 \log_\phi 2}$ then no bad edge will occur more than halfway up the tree.

Consider a message m moving towards the root. We identify two kinds of delays it can encounter, say at vertex u, along the way.

(1) Blocking delay: after m reaches u, it is *not* sent to $f(u)$ at the first opportunity $(u, f(u))$ becomes active. (There are at least p messages in $Q(u)$ ahead of m.)

(2) Coloring delay: after m arrives at u, it is sent to $f(u)$ at the first opportunity $(u, f(u))$ becomes active, but this is *two* rounds after m arrives at u. (This delay of one round is entirely due to the coloring of the edges incident to u.)

The following will be useful.

LEMMA 5. *If $(u, f(u))$ is bad, then the last message u sends $f(u)$, say m_{last}, did not encounter any blocking delay before reaching $f(u)$.*

Proof. Suppose v is a descendant of u at which m_{last} encounters a blocking delay.

CLAIM 1. *$(v, f(v))$ is good.*

Proof. If $(v, f(v))$ were bad, then the last two packets sent upward on $(u, f(u))$ were not full. The first of these packets cleared all messages at v waiting to be sent to v. Since m_{last} is in the following packet sent up by v, it could not have encountered a blocking delay at v, a contradiction. □

CLAIM 2. *If $(v, f(v))$ is good, all the edges in the path from v to $f(u)$ must also be good. In particular, $(u, f(u))$ would be good, a contradiction.*

Proof. Let $(w, f(w))$ be the first bad edge we come to as we move upward from v to $f(u)$. Let x, y denote the sons of w, with m_{last} sent upward along (x, w). Then (y, w) must send its last packet upward to w before x does. Moreover, since (x, w) is good, the upward packet on (x, w) before the one containing m_{last} must be full. It follows easily now that $(w, f(w))$ would be good, a contradiction. This proves Claim 2, and completes the proof of Lemma 5. □

Lemma 5 implies that if $(u, f(u))$ is bad, then all messages in the subtree T_u rooted at u will reach $f(u)$ within at most $2h + 3$ rounds, where h denotes the height of u (the distance from u to a leaf descendant), since the last message of T_u to reach u encounters only coloring delays before arriving at u.

We next establish an upper bound on the number of messages that are sent upward on an edge during round r. This bound will be useful in the sequel.

LEMMA 6. *During round $r \geq 2$, an upward packet on $(u, f(u))$ contains at most $2F_r$ messages, where F_r denotes the rth Fibonacci number, with equality if h (the height of u) is at least r.*

Proof. It is easy to see that during rounds 1, 2, and 3, a unit can send at most 1, 2, and 4 messages, respectively. In round $r \geq 4$, a unit can send at most $\min\{p, |\text{Messages learned in rounds } r-1 \text{ and } r-2|\}$. Thus if M_r denotes the maximum number of messages which can be sent upward on an edge at round r, then M_r satisfies $M_r \leq M_{r-1} + M_{r-2}$, $M_2 = 2$, $M_3 = 4$. Thus, $M_r \leq 2F_r$ for $r \geq 2$. □

Let us assume now that $(u, f(u))$ is bad, and that u has height h. We saw above that u will send to $f(u)$ all the messages in T_u, of which there are $2^{h+1} - 1$, within $2h + 3$ rounds. On the other hand, u could send $f(u)$ at most

$$\frac{\phi^2 p + 1}{\sqrt{5}} + \left\lceil \frac{(2h+3) - \log_\phi p}{3} \right\rceil p$$

messages during the first $2h + 3$ rounds. (Simply note that if $(u, f(u))$ has color c, then by Lemma 6 u could send $f(u)$ at most

$$\sum_{j=0}^{\lfloor(\log_\phi p-c)/3\rfloor} 2F_{3j+c} \le 1 + \sum_{k=1}^{\lfloor\log_\phi p\rfloor} F_k = F_{\lfloor\log_\phi p\rfloor+2} < \frac{1}{\sqrt{5}}\left(\phi^{\lfloor\log_\phi p\rfloor+2} + 1\right) \le \frac{\phi^2 p + 1}{\sqrt{5}}$$

messages during the first $\log_\phi p$ rounds.) But this implies that we must have

$$\frac{\phi^2 p + 1}{\sqrt{5}} + \left\lceil \frac{(2h+3) - \log_\phi p}{3} \right\rceil p > 2^{h+1} - 1 \tag{5}$$

in order to have a bad edge $(u, f(u))$ with u at height h. But if $n \ge 4p^{3/2\log_\phi 2}$, it is easily checked that (5) fails to hold for $h > \frac{1}{2}\log_2 n$. In other words, there will be no bad edges in the top half of the tree. In particular, both edges incident to the root will be good.

The following fact is easy to verify (formally by induction on h): Let u be any node of height h. Then, by the end of round $2h + 3$, either u has sent $f(u)$ a full packet or u has completed sending up packets to $f(u)$.

Let us call a good edge $(u, f(u))$ *complete* if u sends $f(u)$ exactly $2F_r$ messages in every round r during which $(u, f(u))$ is active prior to sending $f(u)$ a first full packet. Otherwise, we call a good edge $(u, f(u))$ *incomplete*.

Proof of Theorem 8. Assume for now that both good edges incident to the root are complete. (We will show in a moment that this is true.) Then the root will learn the cumulative message within $\log_\phi p + \left\lceil \frac{3n}{2p} \right\rceil + O(1)$ rounds. Within $\left\lceil \frac{3n}{2p} \right\rceil$ more rounds, both sons of the root will also know the cumulative message, and so both sons will have learned the cumulative message by round $R + O(1)$, where $R = \frac{3n}{p} + \log_\phi p$. To complete the proof, we will show that each u will learn the cumulative message by round $R + 2(\log_2 n - h) + O(1)$, where h denotes the height of u. The proof is by induction on $\log_2 n - h$; the case $\log_2 n - h = 1$ is given above.

Consider an edge $(u, f(u))$, where u has height h. If $f(u)$ does not always have a full backlog of messages to send u once u has completed sending messages upward, then clearly u will learn the cumulative message at most two rounds after $f(u)$ does. So by the induction hypothesis, u will learn the cumulative message by round $R + 2(\log_2 n - h) + O(1)$. Therefore, we assume in the sequel that u always has a backlog of messages to send u once u completes sending messages up to u.

We now consider three cases.

Case 1. $(u, f(u))$ is complete.

Let u be a descendant of the son r_1 of the root, and for the moment assume $(u, f(u))$ and (r_1, root) are colored the same. Let k_1 (respectively, n_1) denote the number of packets (respectively, messages) which are sent from u to $f(u)$ prior to the first full packet. Then the total number of packets sent across the edge $(u, f(u))$ is

$$k_1 + \left\lceil \frac{2^{h+1} - 1 - n_1}{p} \right\rceil + \left\lceil \frac{n - 2^{h+1}}{p} \right\rceil.$$

On the other hand, the total number of packets sent across (r, root) will be

$$k_1 + \left\lceil \frac{(n-1)/2 - n_1}{p} \right\rceil + \left\lceil \frac{n/2}{p} \right\rceil.$$

These two quantities differ by $O(1)$, and thus u will know the cumulative message by round $R + O(1)$. The argument remains essentially unchanged if $(u, f(u))$ and (r, root) are colored differently.

Case 2. $(u,\ f(u))$ is incomplete.

By the fact given above, u must have sent its first full packet to $f(u)$ by round $2h+3$. Thus the number of rounds before u learns the cumulative message is at most

$$3\left\lceil\frac{2^{h+1}-1}{p}\right\rceil+3\left\lceil\frac{n-2^{h+1}+1}{p}\right\rceil.$$

We would be done if we had

$$R+2(\log_2 n-h)\geq(2h+3)+3\left\lceil\frac{2^{h+1}-1}{p}\right\rceil+3\left\lceil\frac{n-2^{h+1}+1}{p}\right\rceil$$

or $2\log_2 n\geq 4h-\log_\phi p$. However, $h<\log_\phi p$ (or else $2F_h>p$ and u would have sent $2F_r$ messages to $f(u)$ in each round r prior to sending the first full packet, contradicting the fact that $(u,\ f(u))$ is incomplete). Thus, it suffices to have $2\log_2 n\geq 3\log_\phi p$, or $n\geq p^{3/2\log_\phi 2}$, which we have by assumption.

Case 3. $(u,\ f(u))$ is bad or all-partial.

Then u will have completed sending messages up to $f(u)$ within $2h+3$ rounds. So the number of rounds before u learns the cumulative message will be

$$(2h+3)+3\left\lceil\frac{2^{h+1}-1}{p}\right\rceil+O(1).$$

We would be done if we had

$$R+2(\log_2 n-h)\geq 2h+3\left\lceil\frac{n-2^{h+1}}{p}\right\rceil.$$

Since $R\geq\left\lceil\frac{3n}{p}\right\rceil$, it follows easily that the desired inequality will be true if $h\leq\frac{1}{2}\log_2 n$. But if $n\geq 4p^{3/2\log_\phi 2}$ and u has height h, we have argued that $(u,\ f(u))$ cannot be bad if $h>\frac{1}{2}\log_2 n$. The same argument shows that $(u,\ f(u))$ cannot be all-partial if $h>\frac{1}{2}\log_2 n$. Thus $h\leq\frac{1}{2}\log_2 n$ as desired.

This completes the proof of Theorem 8. □

We now establish a lower bound for the number of rounds needed to gossip in B_n using size p packets. It matches the upper bound in Theorem 8 to within $O(1)$ rounds.

THEOREM 9. $g(p,\ B_n)\geq\frac{3n}{p}+2\log_2 n+\log_\phi p-O(1)$.

Proof. In what follows, we assume no vertex receives the same message twice. Clearly this is no loss in generality.

Let x denote a son of the root of B_n, and let T_1, T_2, T_3 denote the binary subtrees of B_n as shown in Fig. 8.

Consider any round r_0 after which x knows $<p$ messages, including his own. We begin with two useful facts concerning round r_0.

(1) After round r_0, x must still receive $>\lceil(n-1)/p\rceil-1$ packets.

Proof of (1). After round r_0, x must still receive

$$\geq\sum_{i=1}^{3}\left\lceil\frac{|\text{Messages in } T_i \text{ not known to } x \text{ after round } r_0|}{p}\right\rceil$$

$$\geq\left\lceil\frac{1}{p}\sum_{i=1}^{3}|\text{Messages in } T_i \text{ not known to } x \text{ after round } r_0|\right\rceil$$

$$>\left\lceil\frac{1}{p}(n-1-p)\right\rceil=\left\lceil\frac{n-1}{p}\right\rceil-1.\qquad\square$$

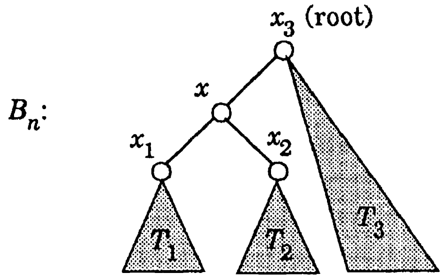


FIG. 8.

(2) After round r_0, x must still send

$$\geq \sum_{i=1}^{3} \left\lceil \frac{n - |T_i|}{p} \right\rceil - 3$$

packets.

Proof of (2). After round r_0, x still needs to send

$$\geq \sum_{i=1}^{3} \left\lceil \frac{|n - |T_i| \text{ messages in } T_i \text{ not known to } x_i \text{ after round } r_0|}{p} \right\rceil$$
$$\geq \sum_{i=1}^{3} \left\lceil \frac{n - |T_i|}{p} \right\rceil - 3 \text{ packets,}$$

since each x_i knows less than p messages outside of T_i after round r_0. □

We observe that

$$\sum_{i=1}^{3} \left\lceil \frac{n - |T_i|}{p} \right\rceil - 3 = 2\left\lceil \frac{3n+3}{4p} \right\rceil + \left\lceil \frac{n-1}{2p} \right\rceil - 3 = \frac{2n}{p} - O(1).$$

Moreover, we note that the last message sent by x must follow the last message received by x. Finally, it can be seen that the last message sent by x needs at least $2\log_2 n - O(1)$ rounds to reach all the leaves of the T_i to which it is sent.

In summary, we see that

$$g(p, B_n) \geq r_0 + \left\lceil \frac{n-1}{p} \right\rceil - 1 + \frac{2n}{p} - O(1) + 2\log_2 n - O(1)$$
$$= r_0 + \frac{3n}{p} + 2\log_2 n - O(1).$$

So, to complete the proof, it suffices to show that there exists a value of r_0 satisfying

(3) $r_0 \geq \log_\phi p - O(1)$.

Proof of (3). To prove (3), let m_r denote the maximum number of messages, including its own, that the root of a binary tree could know after r rounds. It is easy to see that m_r satisfies the recurrence relation $m_r = m_{r-1} + m_{r-2} + 1, m_0 = 1, m_1 = 2$. (Intuitively, one should get as many messages as possible to the jth son of the root, for $j = 1$ and 2, in $r - j$ rounds, and then use round $r - 1$ (respectively, r) to have the second (respectively, first) son send everything it knows to the root. The addition of 1 is for the root's own message.) Solving this recurrence yields $m_r = F_{r+3} - 1$, where F_k is the kth Fibonacci number.

Consider now the earliest round at which x could know $\geq p$ messages, including its own. It is easy to verify that after r rounds, x could know at most $m_{r-4} + m_{r-2} + m_{r-1} + 2 = F_{r-1} + F_{r+1} + F_{r+2} - 1 = 3F_{r+1} - 1$ messages (see Fig. 9).

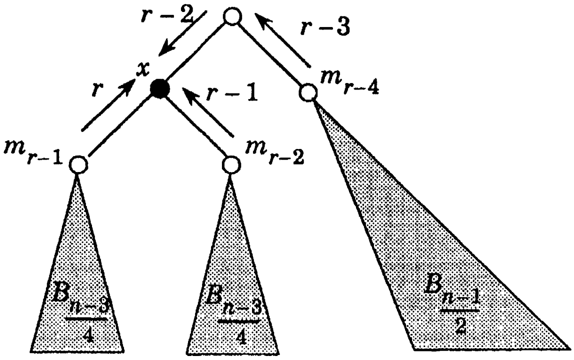


FIG. 9.

Thus, for x to know $\geq p$ messages, we need $\geq r$ rounds, where r satisfies $3F_{r+1} - 1 \geq p$, or $\phi^r \geq F_{r+1} \geq (p+1)/3$, or $r \geq \log_\phi((p+1)/3) \geq \log_\phi p - 3$. Thus, we can take $r_0 \geq \log_\phi p - O(1)$ as asserted. This proves (3), and completes the proof of Theorem 9. □

REFERENCES

[1] A. Bagchi, S. L. Hakimi, J. Mitchem, and E. F. Schmeichel, *Parallel algorithms for gossiping by mail*, Inform. Process. Lett., 34 (1990), pp. 197–202.

[2] A. Bagchi, E. F. Schmeichel, and S. L. Hakimi, *Sequential information dissemination by packets*, Networks, 22 (1992), pp. 317–333.

[3] A. Bagchi and S. L. Hakimi, *Information dissemination in distributed systems with faulty units*, IEEE Trans. Comput., (to appear).

[4] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North Holland, Amsterdam, 1976.

[5] R. Bumby, *A problem with telephones*, SIAM J. Alg. Discrete Math., 2 (1981), pp. 13–18.

[6] R. C. Entringer and P. J. Slater, *Gossip and telegraphs*, J. Franklin Inst., 307 (1979), pp. 353–359.

[7] S. Even and B. Monien, *On the number of rounds necessary to disseminate information*, in Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, 1989.

[8] F. Harary and A. Schwenk, *The communication problem on graphs and digraphs*, J. Franklin Inst., 297 (1974), pp. 491–495.

[9] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman, *A survey of gossiping and broadcasting in communication networks*, Networks, 18 (1988), pp. 319–349.

[10] D. Kleitman and J. B. Shearer, *Further gossip problems*, Discrete Math., 30 (1980), pp. 150–156.

[11] D. W. Krumme, *Fast gossiping for the hypercube*, SIAM J. Comput., 21 (1992), pp. 365–380.

[12] D. W. Krumme, G. Cybenko, and K. N. Venkataraman, *Gossiping in minimal time*, SIAM J. Comput., 21 (1992), pp. 111–139.

[13] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, UK, 1965,

[14] O. Wolfson and A. Segall, *The communication complexity of atomic commitment and of gossiping*, SIAM J. Comput., 20 (1991), pp. 423–450.

SIAM J. COMPUT.
Vol. 23 No. 2, pp. 373–386, April 1994

GENERATING LINEAR EXTENSIONS FAST*

GARA PRUESSE† AND FRANK RUSKEY‡

Abstract. One of the most important sets associated with a poset $\mathcal{P}$ is its set of linear extensions, $E(\mathcal{P})$. This paper presents an algorithm to generate all of the linear extensions of a poset in constant amortized time, that is, in time $O(e(\mathcal{P}))$, where $e(\mathcal{P}) = |E(\mathcal{P})|$. The fastest previously known algorithm for generating the linear extensions of a poset runs in time $O(n \cdot e(\mathcal{P}))$, where n is the number of elements of the poset. The algorithm presented here is the first constant amortized time algorithm for generating a "naturally defined" class of combinatorial objects for which the corresponding counting problem is #P-complete. Furthermore, it is shown that linear extensions can be generated in constant amortized time where each extension differs from its predecessor by one or two adjacent transpositions. The algorithm is practical and can be modified to count linear extensions efficiently and to compute $P(x < y)$, for all pairs x, y, in time $O(n^2 + e(\mathcal{P}))$.

Key words. poset, linear extension, transposition, combinatorial Gray code

AMS subject classifications. 05C45, 06A05, 06A06, 68Q25

1. Introduction.

One definition of the adverb "fast" is "in quick succession" (Webster's Collegiate Dictionary [1]). The purpose of this paper is to show that the linear extensions of a partially ordered set (poset) can be generated fast—so fast, in fact, that no algorithm can be faster, up to constant factors. Furthermore, the constants involved are very small and our algorithms extend the practical range of posets for which extensions can be generated and counted.

Linear extensions are of great interest to computer scientists because of their relation to sorting and scheduling problems. For example, there are many NP-complete one-processor scheduling problems with precedence constraints [13], and one obvious way of solving such problems is by generating all linear extensions of the precedence constraints and picking the best extension. Linear extensions are also of interest to combinatorists because of their relation to counting problems [2], [21]. Our results can be used to generate efficiently the standard Young Tableau of a given shape, alternating permutations, and any of the many other combinatorial objects that can be viewed as linear extensions of particular posets.

Given a poset $\mathcal{P}$, two questions naturally arise. The *generation question* asks whether the linear extensions, $E(\mathcal{P})$, of $\mathcal{P}$ can be efficiently generated. The *counting question* asks whether $e(\mathcal{P})$, the size of the set $E(\mathcal{P})$, can be efficiently determined. The recent result of Brightwell and Winkler [4], that the counting question is #P-complete, indicates that the counting question may be no easier than the generation question. We give the best possible answer to the generation question in the sense that our algorithm generates $E(\mathcal{P})$ in time complexity $O(e(\mathcal{P}))$ (aside from a small amount of preprocessing).

We say that a generation algorithm runs in *constant amortized time* if it runs in time $O(N)$, where N is the number of objects generated. In the case of linear extensions we assume that a unit cost oracle that takes two poset elements a and b and returns whether $a \prec b$ or not is available. We also assume that the poset elements have been labeled in a particular manner, to be described later. This labeling can be carried out in time $O(n^2)$ on an n element poset. Aside from the space used for the poset, the amount of space required by our algorithm is

*Received by the editors July 24, 1991; accepted for publication (in revised form) December 3, 1992.

†Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4, (gara@cs.utoronto.ca). The work of this author was supported in part by Natural Sciences and Engineering Research Council of Canada while the author was visiting Simon Fraser University, Burnaby, Canada.

‡Department of Computer Science, University of Victoria, Victoria, British Columbia, Canada, V8W 2Y2 (fruskey@csr.uvic.ca). The work of this author was supported in part by Natural Sciences and Engineering Research Council of Canada grant A3379.

$O(n)$. No constant amortized time generation algorithm was previously known for a class of combinatorial objects for which the corresponding counting problem is #P-complete.

The problem of generating the linear extensions of a poset has been considered by Knuth and Szwarcfiter [12], Varol and Rotem [24], and Kalvin and Varol [11]. In these papers the term "topological sorting" is used instead of "linear extension." The most efficient of these algorithms appears to be that of Varol and Rotem [24], whose time complexity is given as $O(n \cdot e(\mathcal{P}))$ in [11], where n is the number of elements in the poset. It is worth noting that the Varol and Rotem algorithm is very simple and elegant and quite efficient in practice. The only algorithm of which we are aware for counting linear extensions of arbitrary posets is that of Wells [25], but it appears to be difficult to analyze. For particular classes of posets, such as series-parallel or bounded width, efficient algorithms for counting are known (see, e.g., Bouchitte and Habib [3]).

2. Strategy and definitions. A popular strategy for efficiently generating some set of combinatorial objects is to insist that successive objects in the listing differ by some small and prescribed way. Listings of combinatorial objects that have this property are called (generalized or combinatorial) Gray codes. For example, the binary-reflected Gray codes yield a method for generating all the n-bit strings such that each bit string differs from its predecessor by the flipping of one bit. Gray codes have been found for several classes of combinatorial objects; many of these are described in Wilf [26].

We will regard linear extensions as permutations of the elements of the poset. When generating various classes of permutations, the most common "closeness" criteria is that successive permutations differ by a transposition of two of their elements; sometimes this is further restricted to a transposition of adjacent elements only. The well-known algorithm of Steinhaus [22], Johnson [10], and Trotter [23] provides a Gray code listing of all the $n!$ permutations of n elements where each permutation differs from its predecessor by a transposition of two adjacent elements. Thus we say that the $n!$ permutations can be *generated by* (*adjacent*) *transpositions*. The permutations of an n-set correspond to the linear extensions of the poset that is an n element antichain.

In general, it is not always possible to generate the linear extensions of a poset by transpositions, adjacent or not—for example, the linear extensions of the poset consisting of two nontrivial chains and only if n and m are both odd [5], [6], [16]. Thus, the linear extensions of the poset in Fig. 1 (two 2-element chains) cannot be generated by transpositions. The linear extensions of some classes of posets have been shown to be generable by transpositions (see [18], [15], [20]). It is an open problem to characterize the posets that have this property. Even when the linear extensions of a family of posets can be generated by transpositions, a fast algorithm to perform the generation may not exist.

The basic strategy of our initial algorithm is to generate each linear extension twice, where each extension is flagged plus or minus. The algorithm keeps track of the signs of the extensions and only "outputs" the plus extensions. Thus, in a sense, this algorithm falls into the class of generation algorithms that generate more objects than those that are actually output.

We now introduce our terminology and notation.

A poset (or partially ordered set) $\mathcal{P}$ is a reflexive, transitive, and antisymmetric relation $R(\mathcal{P})$ on a set $S(\mathcal{P})$. An ordered pair $(a, b) \in R(\mathcal{P})$ is denoted $a \preceq_{\mathcal{P}} b$ or, when it will not lead to confusion, simply $a \preceq b$. By $a \prec b$ we mean $a \preceq b$ and $a \neq b$. An element a is *minimal* in $\mathcal{P}$ if there is no element b such that $b \prec a$. Let $\text{Min}(\mathcal{P})$ denote the set of minimal elements of $\mathcal{P}$. If $a \prec b$ and there does not exist a c in $S(\mathcal{P})$ such that $a \prec c \prec b$, then we say that b *covers* a. Let $\text{Cover}(a)$ denote the set of elements that cover a. Elements a and b are said to be *incomparable* if $a \npreceq b$ and $b \npreceq a$. We write $a \parallel b$ to indicate that a and b are incomparable. If no pair of elements of $S(\mathcal{P})$ are incomparable, then $\mathcal{P}$ is a *total ordering*. If $\mathcal{P}$ is a total ordering on $S(\mathcal{P}) = \{x_1, x_2, \ldots, x_n\}$ such that $x_i \prec x_j$ if and only if $i < j$, then we sometimes use $x_1 x_2 \cdots x_n$ to denote $\mathcal{P}$. An *extension* of $\mathcal{P}$ is a poset $\mathcal{Q}$ such that $S(\mathcal{P}) = S(\mathcal{Q})$, and $R(\mathcal{P}) \subseteq R(\mathcal{Q})$. An extension of $\mathcal{P}$ which is a total ordering is called a *linear extension* of $\mathcal{P}$. Let $E(\mathcal{P})$ denote the set of linear extensions of $\mathcal{P}$, and let $e(\mathcal{P})$ denote $|E(\mathcal{P})|$. We let $\pm E(\mathcal{P})$ denote $\{+l, -l \mid l \in E(\mathcal{P})\}$.

The *height*, $h(x)$, of an element x is the average position that it occupies in a linear extension. Thus, a minimum element has height 1, a maximum element has height $|S(\mathcal{P})|$, and if $\mathcal{P}$ is an antichain then all elements have height $(|S(\mathcal{P})| + 1)/2$. The probability that x precedes y is denoted $P(x < y)$; it is the number of extensions in which x precedes y divided by the total number of extensions. In connection with sorting algorithms it is desirable to find pairs of elements x and y where $P(x < y)$ is close to 1/2.

For $T \subseteq S(\mathcal{P})$, we let $\mathcal{P} \backslash T$ denote the poset on the set $S(\mathcal{P}) \backslash T$ with the relations set $R(\mathcal{P}) \cap (S(\mathcal{P}) \backslash T)^2$. Suppose a and b are incomparable elements of $S(\mathcal{P})$ such that a has the same relationship to all other elements of $S(\mathcal{P})$ as b; more precisely, suppose that, for all $c \in S(\mathcal{P})$, $c \prec a$ if and only if $c \prec b$ and $a \prec c$ if and only if $b \prec c$. Then a and b are called *siblings*. For posets $\mathcal{P}$ and $\mathcal{Q}$, if $R(\mathcal{P}) \cup R(\mathcal{Q})$ is antisymmetric, then we let $\mathcal{P} + \mathcal{Q}$ denote the poset on the set $S(\mathcal{P}) \cup S(\mathcal{Q})$ with the relation set that is the transitive closure of $R(\mathcal{P}) \cup R(\mathcal{Q})$. For example, $\mathcal{P} + abc$ is the poset on the set $S(\mathcal{P}) \cup \{a, b, c\}$ with the relation set that is the transitive closure of $R(\mathcal{P}) \cup \{(a, b), (b, c)\}$. If $\mathcal{P} + \mathcal{Q} = \mathcal{P}$, then we say $\mathcal{P}$ *induces* $\mathcal{Q}$. For example, if $\mathcal{P} + abc = \mathcal{P}$, then $\{(a, b), (b, c)\} \subseteq R(\mathcal{P})$, and every linear extension of $\mathcal{P}$ has $a \prec b \prec c$; therefore, we say $\mathcal{P}$ induces abc. For element disjoint total orders $\alpha, \beta, \gamma, \delta$, we let $\alpha(\beta + \gamma)\delta$ denote the poset $\alpha\beta\delta + \alpha\gamma\delta$.

Consider the graph that has $E(\mathcal{P})$ as its vertex set, such that two vertices are adjacent in the graph whenever the corresponding linear extensions differ by a single transposition. This graph is called the *transposition graph* of the poset $\mathcal{P}$ and is denoted $G(\mathcal{P})$. The subgraph of $G(\mathcal{P})$ on the same vertex set but containing only the edges that correspond to adjacent transpositions is called the *adjacent transposition graph* and is denoted $G'(\mathcal{P})$. Generating the linear extensions of $\mathcal{P}$ by (adjacent) transpositions is equivalent to finding a Hamiltonian path in the graph $G(\mathcal{P})$ $(G'(\mathcal{P}))$. Figure 1 shows a poset and its transposition graph. If α and β are linear extensions of $\mathcal{P}$, then by $D(\alpha, \beta)$ we denote the distance in $G(\mathcal{P})$ from α to β and by $D'(\alpha, \beta)$ we denote the corresponding distance in $G'(\mathcal{P})$.

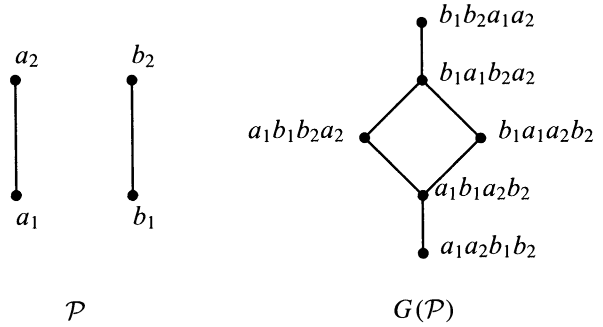


FIG. 1. *A poset and its transposition graph.*

Transposition graphs are bipartite and connected. If the partite sets of $G(\mathcal{P})$ are not the same size, then there is no Hamiltonian cycle through the graph; if the difference in the size of the partite sets is more than one, there is no Hamiltonian path through the graph and thus the linear extensions of $\mathcal{P}$ cannot be generated by transpositions. Ruskey [17] conjectures that this necessary condition for the existence of a Hamiltonian path is also sufficient, suggesting a possible characterization of the posets whose linear extensions can be generated by transpositions. In Fig. 1, the partite sets have a size difference of two, so the linear extensions of that poset cannot be generated by transpositions.

If G is a graph, then let $G \times K_2$ be the graph that results from taking two copies of G and adding the edges that correspond to an isomorphism between the two copies. To differentiate between the copies of G, we will prefix the vertices of one copy of G with "+" and the other with "$-$". For example, Fig. 2 shows $G(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the poset shown in Fig. 1.

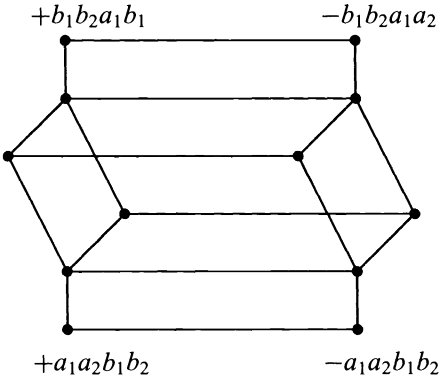


FIG. 2. *The graph* $G(\mathcal{P}) \times K_2$.

A lemma that will be useful in later sections can be stated as follows.

LEMMA 2.1. *If a and b are siblings in* $\mathcal{P}$, *then* $G(\mathcal{P}) \cong G(\mathcal{P} + ab) \times K_2$.

Proof. Observe that $E(\mathcal{P}) = E(\mathcal{P} + ab) \cup E(\mathcal{P} + ba)$. For any linear extension l of $\mathcal{P}$, transposing a and b in l yields another linear extension of $\mathcal{P}$. Therefore, the operation that transposes a and b in a linear extension provides an isomorphism between $G(\mathcal{P} + ab)$ and $G(\mathcal{P} + ba)$. □

If $e(\mathcal{P}) = 1$ (i.e., if $\mathcal{P}$ is a total order), $G(\mathcal{P}) \times K_2$ is an edge. For the purpose of inductively showing the existence of Hamiltonian cycles, we consider this graph to have a Hamiltonian cycle, since it has a Hamiltonian path such that the endpoints are adjacent.

3. The graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian. The proof that $G'(\mathcal{P}) \times K_2$ is Hamiltonian forms the basis of the efficient algorithm to be presented in the next section. That this is true for a certain kind of poset, called a B-poset, was shown by Pruesse and Ruskey [15], and this result will be used in the proof of the general case.

DEFINITION 3.1. A *B-poset* is a poset $\mathcal{P}$ whose elements can be partitioned into two disjoint chains, $x_1 \prec x_2 \prec \cdots \prec x_n$ and $y_1 \prec y_2 \prec \cdots \prec y_m$, such that $y_j \not\preceq x_i$ for all i and j, where $1 \le i \le n$ and $1 \le j \le m$.

An example of a B-poset is shown in Fig. 3. Note that $\kappa = x_1x_2\cdots x_ny_1y_2\cdots y_m$ is a linear extension of any B-poset. The extension κ is called the *canonical linear extension* of a B-poset. Define $mr(x_i)$ to be the largest index j such that $x_i \parallel y_j$; if $x_i \prec y_1$, then $mr(x_i) = 0$. For the B-poset of Fig. 3, $mr(a) = 4$ and $mr(b) = 7$.

The following lemma was proved in [15].

LEMMA 3.2. *Let* $\mathcal{P}$ *be a B-poset. Then there exists a Hamiltonian cycle in* $G'(\mathcal{P}) \times K_2$ *that uses the edge* $[+\kappa, -\kappa]$.

Figure 4 shows the graph $G'(\mathcal{P}) \times K_2$, where $\mathcal{P}$ is the B-poset shown in Fig. 3. The edges corresponding to the isomorphism between the two copies of $G'(\mathcal{P})$ have been omitted for the

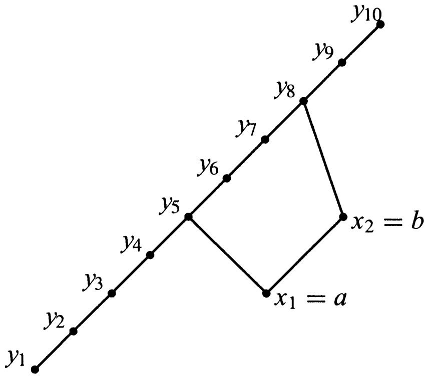


FIG. 3. *A B-poset.*

sake of clarity. One can think of traveling up a vertical edge as transposing $b = x_2$ with its neighbor on the right and traveling along a horizontal edge as transposing $a = x_1$ with one of its neighbors. A Hamiltonian path between $+\kappa$ and $-\kappa$ is shown in Fig. 5. All B-posets used in the remainder of the paper have $n = 2$; we call these *2B-posets.*

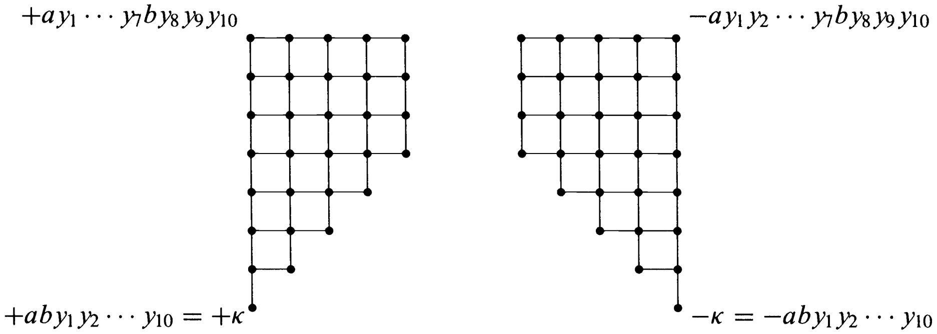

$+ay_1 \cdots y_7by_8y_9y_{10}$
$-ay_1y_2 \cdots y_7by_8y_9y_{10}$
$+aby_1y_2 \cdots y_{10} = +\kappa$
$-\kappa = -aby_1y_2 \cdots y_{10}$

FIG. 4. *The graph $G'(\mathcal{P}) \times K_2$.*

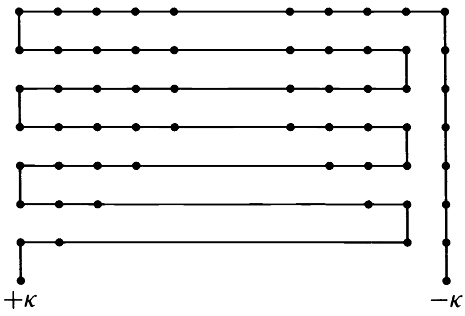


FIG. 5. *A Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$.*

A graph similar to that shown in Fig. 4 arises whenever $a \parallel y_1$; we call this the *typical case.* If $a \prec y_1$ then $G'(\mathcal{P})$ is a path; we call this the *atypical case.* In other words, the typical case occurs when $mr(a) > 0$ and the atypical case occurs when $mr(a) = 0$.

THEOREM 3.3. *For every poset $\mathcal{P}$, the graph $G'(\mathcal{P}) \times K_2$ is Hamiltonian.*

Proof. The proof of the theorem is by induction on $|S(\mathcal{P})|$. For the base cases of the induction, $\mathcal{P}$ is the poset on zero or one element; in both of these cases $G'(\mathcal{P}) \times K_2$ is an edge.

Suppose $|S(\mathcal{P})| > 1$. If $\mathcal{P}$ has a unique minimum a, then $G'(\mathcal{P}) \cong G'(\mathcal{P}\backslash\{a\})$, and by the inductive hypothesis $G'(\mathcal{P}\backslash\{a\}) \times K_2$ is Hamiltonian.

Otherwise, let $\mathcal{P}$ have two minimal elements a and b. By the inductive hypothesis, the graph $G'(\mathcal{P}\backslash\{a, b\}) \times K_2$ has a Hamiltonian cycle H'. Replace each signed linear extension $+\alpha_i$ on H' with $ab\alpha_i$; replace each linear extension $-\alpha_i$ with $ba\alpha_i$. The result is a cycle $\beta_1, \beta_2, \dots, \beta_M$, where $M = 2 \cdot e(\mathcal{P}\backslash\{a, b\})$, in $G'(\mathcal{P}) \times K_2$, which visits exactly those linear extensions in which a and b precede all other elements of $S(\mathcal{P})$. That is, this cycle visits all the linear extensions of

$$\mathcal{P}' = \mathcal{P} + \sum_{x \in S(\mathcal{P})\backslash\{a,b\}} ax + bx.$$

The poset $\mathcal{P}'$ is $\mathcal{P}$ extended so that a and b are covered by every other element of $\operatorname{Min}(\mathcal{P})$.

For each $\beta_i = x_i y_i \zeta_i$, where $x_i = a$, $y_i = b$ or $x_i = b$, $y_i = a$, the poset $\mathcal{P} + x_i y_i + \zeta_i$ is a B-poset. By Lemma 3.2, there is a Hamiltonian path through $G'(\mathcal{P} + x_i y_i + \zeta_i) \times K_2$ from $+\beta_i$ to $-\beta_i$. We substitute the occurrence of β_i with this path in H', for each odd i. For each even i, we substitute the reverse of this path for the occurrence of β_i. Call the resulting walk H.

To prove that H is a Hamiltonian cycle through $G'(\mathcal{P}) \times K_2$, it is necessary to show that every vertex on the cycle H is a linear extension of $\mathcal{P}$; this is true, since $E(\mathcal{P}') \subseteq E(\mathcal{P})$, and hence each B-poset generated is an extension of $\mathcal{P}$. It is also necessary to show that for each linear extension l of $\mathcal{P}$, $+l$ and $-l$ both occur exactly once on H. Suppose l induces the order xy on $\{a, b\}$ and the order ζ on $S(\mathcal{P})\backslash\{a, b\}$. Then $xy\zeta$ is a linear extension of $\mathcal{P}'$, and l is a linear extension of the B-poset $\mathcal{P} + xy + \zeta$; also, every other B-poset that is generated either does not induce the order xy or does not induce the order ζ. Therefore, $+l$ and $-l$ are generated only during the generation of $E(\mathcal{P} + xy + \zeta)$; i.e., each $+l$ and $-l$ are generated exactly once. □

Observe that the reference to Lemma 3.2 in the preceding proof was not strictly necessary because the B-posets that occur are all 2B-posets. In the "typical" case, the cycle of Fig. 5 could be used; in the "atypical" case the cycle is obvious (move b to the right as far as possible, change signs, and then move the b back to the left). If $mr(b)$ is even, the cycle of Fig. 5 is slightly different and uses the edge $[+a\gamma b, -a\gamma b]$, where $\gamma = y_1 y_2 \cdots y_m$. These cycles are used in the algorithm of the following section.

COROLLARY 3.4. *If $\mathcal{P}$ is a poset with a pair of siblings, then $G(\mathcal{P})$ is Hamiltonian.*

Proof. Suppose $\mathcal{P}$ has a pair of siblings a, b. By Theorem 3.3, $G'(\mathcal{P} + ab) \times K_2$ is Hamiltonian; therefore, $G(\mathcal{P} + ab) \times K_2$ is Hamiltonian. Hence by Lemma 2.1, $G(\mathcal{P})$ is Hamiltonian. □

It is an open problem to determine whether $G'(\mathcal{P})$ is Hamiltonian, where $\mathcal{P}$ is a poset with a pair of siblings.

4. The algorithm. The proof of Theorem 3.3 is constructive. In this section, we present the recursive algorithm implicit in the inductive proof. The algorithm runs in *constant amortized time*, i.e., generating all the linear extensions of a poset $\mathcal{P}$ takes time $O(e(\mathcal{P}))$.

We first give an overview of the algorithm and use a small example to give a general description of how it works. We then give the details of the algorithm and a proof of its correctness.

The algorithm is an *in-place* algorithm; it maintains an array `le`, which contains the current linear extension, and maintains a variable `IsPlus`, which keeps track of the sign

("+" or "−"). We go from one linear extension to the next by making changes to the array or reversing the sign.

The main procedure used by the algorithm, which we call GenLE, is recursive and basically follows the path indicated in Fig. 5. Every level of the recursion has an associated pair of minimal elements of the current subposet. For example, in the poset shown in Fig. 1, a_1, b_1 are a pair of minimal elements of $\mathcal{P}_1 = \mathcal{P}$ and a_2, b_2 are a pair of minimal elements of $\mathcal{P}_2 = \mathcal{P}_1 \backslash \{a_1, b_1\}$. These pairs are determined by some preprocessing, which will be described later.

The procedures Move and Switch are used to change the current linear extension. They operate in $O(1)$ time as follows.

Switch(i): If $i = 0$, then the sign is changed; that is, IsPlus is changed. If $i > 0$, then a_i and b_i are transposed.

Move(x,left): This call transposes x with the element on its left.

Move(x,right): This call transposes x with the element on its right.

Each time a new linear extension l of $\mathcal{P}_i$ is generated by the call GenLE(i) (i.e., each time Move or Switch is called), GenLE(i-1) is called; the call GenLE(i-1) moves $a_1, b_1, \ldots, a_{i-1}, b_{i-1}$ in all possible ways through l, while maintaining the order $a_{i-1} \prec b_{i-1}$ (or $b_{i-1} \prec a_{i-1}$, depending on their order at the point of calling GenLE(i-1)). If $i = 1$, then GenLE(i-1) does nothing.

For example, starting with $+a_1b_1a_2b_2$ and executing the calling sequence GenLE(2), Switch(2), GenLE(2) on the poset shown in Fig. 1 leads to the trace of calls shown in Fig. 6.

Procedure Call	**Linear Extension**
	$+a_1b_1a_2b_2$
GenLE(2)	
GenLE(1)	
Move($b_1, \rightarrow$)	$+a_1a_2b_1b_2$
Switch(0)	$-a_1a_2b_1b_2$
Move($b_1, \leftarrow$)	$-a_1b_1a_2b_2$
Switch(1)	$-b_1a_1a_2b_2$
GenLE(1)	
Switch(0)	$+b_1a_1a_2b_2$
Switch(2)	$+b_1a_1b_2a_2$
GenLE(2)	
GenLE(1)	
Move($a_1, \rightarrow$)	$+b_1b_2a_1a_2$
Switch(0)	$-b_1b_2a_1a_2$
Move($a_1, \leftarrow$)	$-b_1a_1b_2a_2$
Switch(1)	$-a_1b_1b_2a_2$
GenLE(1)	
Switch(0)	$+a_1b_1b_2a_2$

FIG. 6. *The trace of the calling sequence for the poset of Fig.* 1.

We now follow with the details of our implementation. The reader should refer to the Pascal procedure GenLE of Fig. 8.

The implementation of the algorithm maintains four global arrays: array le is the linear extension; array li is its inverse. Arrays a and b store the elements a_i and b_i. In our discussion of the algorithm, a_i and b_i are considered fixed at the outset and unchanging throughout the run of the algorithm. The arrays will be maintained so that a[i] always contains the value of the leftmost of the ith pair and `b[i]` contains the rightmost. Also, the current sign, either plus (+) or minus (−), is maintained.

The boolean function `Right(x)` is used to determine whether the element `x` can move to the right. It operates in $O(1)$ time as follows:

`Right(b[i])`: Returns true only if `b[i]` is incomparable with the element to its right in the array `le`.

`Right(a[i])`: Returns true only if `a[i]` is incomparable with the element to its right in the array `le` and the element to the right is not `b[i]`.

We now describe our preprocessing. We successively strip off pairs a_i, b_i of minimal elements for $i = 1, 2, \ldots$ until there are no elements left. If a unique minimum element is encountered then it is simply deleted and does not become part of a pair. Let `MaxPair` be the index of the last pair of minimal elements we strip from $\mathcal{P}$, the remainder of $\mathcal{P}$ being a total ordering or empty. This preprocessing is detailed in Fig. 7. Note that `MaxPair` is not uniquely determined by the poset but that it depends on the order in which the elements are stripped from $\mathcal{P}$.

```
$i \leftarrow j \leftarrow 0$;
$\mathcal{Q} \leftarrow \mathcal{P}$;
while $S(\mathcal{Q}) \neq \emptyset$ do
    if $\mathcal{Q}$ has exactly one minimal element $x$ then begin
        $j \leftarrow j + 1$;
        $le[j] \leftarrow x$;
        $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{x\}$;
    end else begin
        let $a', b'$ be any two minimal elements of $\mathcal{Q}$;
        $i \leftarrow i + 1$;
        $j \leftarrow j + 2$;
        $a[i] \leftarrow a'$;
        $b[i] \leftarrow b'$;
        $le[j-1] \leftarrow a'$;
        $le[j] \leftarrow b'$;
        $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{a', b'\}$;
    end {else}
end {while}
MaxPair $\leftarrow i$;
```

FIG. 7. *The preprocessing routine.*

We say the linear extension l is in *proper order up to i* if for all $1 \leq j \leq i$ the elements a_j and b_j are adjacent in l and if l induces the orders $a_1 a_2 \cdots a_i a_h$ and $a_1 a_2 \cdots a_i b_h$ for all h, where $i < h \leq$ `MaxPair`. The initial linear extension of the listing must be properly ordered up to `MaxPair`; the preprocessing of Fig. 7 does this.

Assuming that `Right( b[MaxPair+1] )` is false, the initial call is simply `GenLE( MaxPair + 1 )`; this is the same as the following procedure calls, which we call the *calling sequence.*

```
GenLE( MaxPair ); Switch( MaxPair ); GenLE( MaxPair );
```

The algorithm consists of executing the preprocessing, setting `IsPlus` to plus (+), and then executing the calling sequence.

A Pascal procedure implementing `GenLE` is given in Fig. 8. We now prove the following theorem.

THEOREM 4.1. *The algorithm* `GenLE` *generates the linear extensions along a Hamiltonian path in* $G'(\mathcal{P}) \times K_2$.

```
procedure GenLE ( i : integer );
var mrb,mra,mla,x : integer;  typical : boolean;
begin
if i > 0 then begin
  GenLE( i-1 );
  mrb := 0;   typical := false;
  while Right( b[i] ) do begin
     mrb := mrb + 1;
     Move( b[i], riht );  GenLE( i-1 );
     mra := 0;
     if Right( a[i] ) then begin
        typical := true;
        repeat
           mra := mra + 1;
           Move( a[i], riht );  GenLE( i-1 );
        until not Right( a[i] );
     end {if};
     if typical then begin
        Switch( i-1 );  GenLE( i-1 );
        if odd( mrb ) then mla := mra - 1 else mla := mra + 1;
        for x := 1 to mla do begin
           Move( a[i], left );  GenLE( i-1 );
        end;
     end {if};
  end {while};
  if typical and odd( mrb )
     then Move( a[i], left )
     else Switch( i-1 );
  GenLE( i-1 );
  for x := 1 to mrb do begin
     Move( b[i], left );  GenLE( i-1 );
  end;
end {if};
end {GenLE};
```

FIG. 8. *Pascal procedure* GenLE.

Proof. In order to prove the theorem, we first prove the following proposition.

PROPOSITION 4.2. *Let the linear extension in array* `le` *be* $\xi = \delta a_i b_i \gamma$, *and let* ξ *be properly ordered up to* i. *Then for each linear extension* $l \in E(\mathcal{P} + a_i b_i + \gamma)$, `GenLE(i)` *generates each of* $+l$, $-l$ *exactly once. Furthermore, if* $i = 1$ *then the last extension generated is* $-\xi$, *and if* $i > 1$ *then the last extension generated is* $+\xi'$, *where* ξ' *differs from* ξ *by a transposition of* a_{i-1} *and* b_{i-1}.

Proof. The proof proceeds by induction on i. If $i = 1$, the recursive calls `GenLE(0)` do nothing, `Switch(0)` just changes the sign, and δa_1 is induced by $\mathcal{P}$.

It is easy to confirm that the algorithm in Fig. 8, when stripped of its recursive calls and in which `Switch` just changes the sign, simply follows the path indicated in Fig. 2. In this case, `GenLE(1)` just finds a Hamiltonian path from $+\xi$ to $-\xi$ through $G'(\mathcal{Q}) \times K_2$, where $\mathcal{Q}$ is the 2B-poset $\mathcal{P} + a_1 b_1 + \gamma$.

If $i > 1$, then assume without loss of generality that the sign in storage when `GenLE` is invoked is "+". There are α, β, γ such that $+\xi = +\alpha a_{i-1} b_{i-1} \beta a_i b_i \gamma$. Because of the way the preprocessing selects the pairs a_j, b_j, we are assured that $\mathcal{P}$ induces the order $\beta(a_i + b_i)$ (of course, β could be empty).

As mentioned before, the basic structure of the algorithm, stripped of recursive calls, follows the Hamiltonian path in a 2B-poset as indicated in Fig. 2, where `Switch` now transposes a_{i-1} and b_{i-1}; therefore, it generates

$$+E' = +E(\mathcal{P} + \alpha(a_{i-1} + b_{i-1})\beta(a_i b_i + \gamma)).$$

Each linear extension in $+E'$ is properly ordered up to $i - 1$.

As each such linear extension $+l = +\alpha a_{i-1} b_{i-1} \beta\zeta$ (or $+l' = +\alpha b_{i-1} a_{i-1}\beta\zeta$), where $\zeta \in E(a_i b_i + \gamma)$, is generated, GenLE(i-1) is called on $+l$. By the inductive hypothesis, that call generates $\pm E(\mathcal{P} + a_{i-1}b_{i-1} + \beta\zeta)$ (or $\pm E(\mathcal{P} + b_{i-1}a_{i-1} + \beta\zeta)$, respectively), starting at $+l$ and ending at $+l'$ if $i > 2$, and ending at $-l$ if $i = 2$. Since there are an even number of vertices in the product of the graph with an edge, there are an even number of calls to GenLE(i-1). Thus if $i > 2$ then the sign of the final permutation is unchanged, and if $i = 2$ then the relative ordering of a_{i-1} and b_{i-1} is unchanged. The union over all such ζ is

$$\pm E(\mathcal{P} + a_{i-1}b_{i-1} + a_i b_i + \gamma) \cup \pm E(\mathcal{P} + b_{i-1}a_{i-1} + a_i b_i + \gamma) = \pm E(\mathcal{P} + a_i b_i + \gamma). \qquad \square$$

Let a, b be a[MaxPair], b[MaxPair] respectively, and suppose the calling sequence is executed on the preprocessed poset $\mathcal{P}$. By the proposition, the first call to GenLE generates $\pm E(\mathcal{P} + ab)$; then a and b are transposed, and then $\pm E(\mathcal{P} + ba)$ is generated. Therefore, $E(\mathcal{P})$ is generated, and the theorem is proved. □

In analyzing the time complexity of the algorithm, we assume that Right, Switch, and Move can be implemented in constant time. This is easily accomplished as long as the inverse li of le is maintained. Each call to Move and Switch generates one more linear extension. Observe that the call GenLE(i) generates at least two calls to GenLE(i-1). Each iteration of a while-loop or for-loop in the algorithm executes a Move, thereby generating a linear extension. The only occasion in which GenLE can be recursively called and no linear extension generated is when $i = 0$, and this happens at most once per linear extension generated. Therefore, the algorithm runs in constant time per linear extension, when generating $\pm E(\mathcal{P})$. By suppressing the linear extensions which are prefixed with "$-$", we generate $E(\mathcal{P})$ in constant amortized time. Another way to think of the preceding argument is to consider the underlying computation tree, where each internal node is a recursive call and each leaf is a linear extension. The total amount of computation can be apportioned so that each node is assigned a constant amount of computation. Since each internal node has at least two children, the number of leaves is greater than the number of internal nodes and, therefore, the total amount of computation is proportional to the number of leaves.

Observe that the generation of the minus ("$-$") vertices only occurs when $i = 1$ in Algorithm GenLE. This suggests that $i = 1$ be treated as a special case and that minus ("$-$") vertices be omitted entirely by simply skipping to the next plus ("+") vertex. If this is done, then it saves some computation, but the same list of extensions is produced as before, and successive extensions can differ by a large number of transpositions.

If one only wants to compute the number of extensions, then some computation can be saved by only computing the number of vertices at the $i = 1$ level of the recursion, and not generating the extensions explicitly, i.e., never moving a_1 and b_1. The number of vertices (extensions) can be determined from $mr(a) =$ mra and $mr(b) =$ mrb; the number is $(mr(a) + 1)[mr(b) + 1 - mr(a)/2]$. Furthermore, $mr(a)$ and $mr(b)$ change by at most unity from one extension to the next, since only adjacent transpositions are used. This leads to an algorithm whose running time is $O(e(\mathcal{P} \setminus \{a_1, b_1\}))$. In general, we have

$$2 \cdot e(\mathcal{P} \setminus \{a_1, b_1\}) \leq e(\mathcal{P}) \leq n(n-1) \cdot e(\mathcal{P} \setminus \{a_1, b_1\}).$$

The lower bound is attained when $a_1 \prec c$ and $b_1 \prec c$ for all elements c of $\mathcal{P} \setminus \{a_1, b_1\}$. The upper bound is attained when a_1 and b_1 are maximal, as well as minimal.

5. Gray codes for linear extensions. We now show that linear extensions can be listed so that successive extensions differ by at most a few adjacent transpositions. We first show the existence of such listings and then how to modify the results of the previous sections to show that the set of linear extensions of a poset can be listed so that successive extensions differ by only one or two adjacent transpositions. Let us say that an ordering $\alpha_1, \alpha_2, \dots, \alpha_{e(\mathcal{P})}$ of the extensions of $\mathcal{P}$ has *delay C* if $D'(\alpha_i, \alpha_{i+1}) \le C$ for all $0 \le i < e(\mathcal{P})$, where $\alpha_0 = \alpha_{e(\mathcal{P})}$. Thus we are going to show the existence of a delay 2 ordering of $E(\mathcal{P})$. Furthermore, such a listing can be done in constant amortized time. The existence of a delay 3 ordering is not difficult to show.

If G is a graph then by G^k we denote the graph with the same vertex set as G but which has an edge between every pair of vertices that are connected by a path of length at most k in G. In other words, if M is the incidence matrix of G, then M^k is the incidence matrix of G^k, where arithmetic is done mod 2. The *cube* of G is G^3, and the *square* of G is G^2. A poset $\mathcal{P}$ has a delay k ordering if and only if $G'(\mathcal{P})^k$ is Hamiltonian. A result of Sekanina [19] is that the cube of every connected graph is Hamiltonian. Since $G'(\mathcal{P})$ is always connected, $G'(\mathcal{P})^3$ is Hamiltonian and a delay 3 ordering exists.

The graph $G'(\mathcal{P})$ is not always 2-connected; otherwise the existence of a delay 2 ordering would by implied by a result of Fleischner [7], which states that the square of every 2-connected graph is Hamiltonian.

Even though $G'(\mathcal{P})$ is not in general 2-connected, the posets with 2-connected transposition graphs are easy to characterize. First, consider the question of which transposition graphs have pendant vertices. If $\mathcal{P}$ consists of two disjoint chains, then $G'(\mathcal{P})$ has two pendant vertices; if $\mathcal{P}$ is a B-poset and is not the disjoint union of two chains, then $G'(\mathcal{P})$ has one pendant vertex; otherwise, $G'(\mathcal{P})$ has no pendant vertices.

LEMMA 5.1. *For every poset $\mathcal{P}$, the graph $H(\mathcal{P})$ is 2-connected, where $H(\mathcal{P})$ is $G'(\mathcal{P})$ minus any pendant vertices.*

This may be proved by showing that every pair of incident edges of $H(\mathcal{P})$ is on a 4-, 6-, or 8-cycle. This lemma does not help us in finding an efficient algorithm for listing a delay 2 ordering of $E(\mathcal{P})$. Instead, we prove it by applying Theorem 3.3 and the following lemma.

LEMMA 5.2. *If G is bipartite and $G \times K_2$ is Hamiltonian, then G^2 is Hamiltonian.*

Proof. Let G be a bipartite graph on n vertices, and let

$$(v_1, x_1), (v_2, x_2), \dots, (v_{2n}, x_{2n})$$

be a Hamiltonian cycle through $G \times K_2$, where $v_i \in V(G)$ and $x_i \in V(K_2) = \{1, 2\}$, for all i, $1 \le i \le 2n$. Consider the sequence of vertices $S = v_2, v_4, \dots, v_{2n}$.

Since G is bipartite, so is $G \times K_2$; thus the vertices of S are all the vertices of one partite set of $G \times K_2$. Also, for a vertex u of G, $(u, 1)$ and $(u, 2)$ are adjacent and therefore are in different partite sets of $G \times K_2$. Therefore, each vertex of G appears exactly once in S. For each i, the vertices v_i and v_{i+2} are either of distance one in G (if $x_i \ne x_{i+2}$) or of distance two in G (if $x_i = x_{i+2}$). Therefore, S is a Hamiltonian cycle in G^2. □

We conjecture that Lemma 5.2 may be extended to graphs that are not bipartite.

CONJECTURE 5.3. *If $G \times K_2$ is Hamiltonian, then G^2 is Hamiltonian.*

A graph is in class $\mathcal{H}(s, t)$ if it has a closed walk that visits every vertex at least s times and at most t times. (See [8], [14].) Thus $\mathcal{H}(1, 1)$ is the class of Hamiltonian graphs. Observe that if $G \times K_2$ is Hamiltonian, then $G \in \mathcal{H}(1, 2)$ (just consider the walk that results when the two copies of G are identified). The converse is not true; if G is the triangle with a pendant edge added to each vertex, then $G \in \mathcal{H}(1, 2)$ but $G \times K_2$ is not Hamiltonian. Theorem 3.3

shows that for every poset $\mathcal{P}$, the graph $G'(\mathcal{P})$ is in $\mathcal{H}(1, 2)$. In general, these graphs are not Hamiltonian. We make a conjecture.

CONJECTURE 5.4. *If $G \in \mathcal{H}(1, 2)$, then G^2 is Hamiltonian.*

The example of $K_{2,6}$ shows that the converse of the conjecture is false.

The proof of Lemma 5.2 is constructive; applying that construction to $G'(\mathcal{P})$ yields the following result.

THEOREM 5.5. *The linear extensions of any poset can be generated with delay* 2 *in constant amortized time.*

Proof. We run the Algorithm `Genle` given in Fig. 8, but instead of suppressing the linear extensions with a negative sign we suppress every other linear extension; i.e., if we generate the list $l_1, l_2, l_3, l_4, l_5, \ldots,$ then we output the list $l_1, l_3, l_5, \ldots.$ By the proof of Lemma 5.2, this is a delay 2 listing of the linear extensions. It has the same running time as `Genle`, i.e., constant amortized time. □

In the remainder of this section we discuss how to use the algorithm to compute $P(x < y)$ and $h(x)$. We use the version of `GenLE` that generates each extension exactly twice, where each successive extension differs by an adjacent transposition from its predecessor. We first discuss how to compute $P(x < y)$.

Let us define an xy-run to be a maximal sequence of successive extensions where x precedes y. We maintain two arrays of integers, call them S and T. The value of $S[x, y]$ is the sum of the lengths of the previous xy-runs. The value of $T[x, y]$ is the iteration at which the current xy-run started. At each iteration, exactly one adjacent pair, say xy, is transposed. If this occurs at the tth iteration, then $S[x, y]$ is incremented by $t - T[x, y]$ and $T[y, x]$ is set to t. At the termination of the algorithm, $P(x < y)$ is $S[x, y]$ divided by $2e(\mathcal{P})$. Since only a constant amount of update is done at each iteration, the total computation is $O(n^2 + e(\mathcal{P}))$.

To compute $h(x)$ we proceed in a similar fashion. An x-run is a maximal sequence of extensions in which x occupies the same position. Here the value of $S[x]$ is the weighted sum of the lengths of the previous x-runs and $T[x]$ is the iteration at which the current x-run started. At each iteration exactly one adjacent pair, say xy, is transposed. If this occurs at the tth iteration, then for $z = x, y$, $S[z]$ is incremented by $p[z] * (t - T[z])$ and $T[z]$ is set to t, where $p[z]$ is the position that z occupied in the extension. At termination the value of $h(x)$ is $S[x]$ divided by $2e(\mathcal{P})$.

6. Concluding remarks.

The algorithm given in §6 generates the linear extensions of a poset in constant amortized time, an improvement over the $O(n)$ amortized time algorithm of [11], which is the fastest previously known algorithm. A further refinement to the work presented here would be to generate the linear extensions by a "loopfree" algorithm (i.e., constant computation in the worst case in producing a new extension from the current one).

We have fully implemented the counting and generating algorithms in C and found them to be quite efficient in practice. On a Sun SPARCstation SLC, the program generated the 2,702,765 extensions of a 12-element fence poset in 4.2 seconds. These extensions are counted by the Euler numbers; counting took 1.7 seconds. The 199,360,981 extensions of a 14-element fence were counted in 91 seconds and generated in 281 seconds. The 2,674,440 extensions of the 2 by 14 grid were generated in 8.3 seconds. These extensions are counted by the Catalan numbers.

Enumeration of the linear extensions of a poset has recently been shown to be #P-complete. The algorithm in §3 constitutes the first constant amortized time algorithm for generating a naturally defined class of combinatorial objects where the associated counting problem is #P-complete. This leads to some interesting questions about the complexity of generating other combinatorial objects for which counting is #P-complete.

Do all #P-complete problems admit constant amortized time generation algorithms? In asking this question we assume that an initial object has been supplied as part of the input. For example, for Hamiltonian cycles the input would consist of a graph G and a Hamiltonian cycle in G; similarly, in this paper we have assumed that $12 \cdots n$ is a linear extension of the input poset.

Intuitively, if the existence question is difficult (NP-complete), then the generation question will be difficult as well, but we have not answered, or even formalized, this intuition. Even if the existence question is easy (in P), there are many problems for which the complexity of generating is unknown. For example, can the ideals of a given input poset be generated in constant amortized time? Counting ideals is #P-complete, but finding an ideal is trivial. What about generating the spanning trees of a graph in constant amortized time? Finding a spanning tree and counting the number of such trees are both in P, but no one has discovered a constant amortized time algorithm for generating the spanning trees. Is there, in fact, any interesting relationship between constant amortized time generation algorithms and the complexity of existence and/or counting? Some related questions, for various polynomial time complexity measures (instead of constant amortized time), were considered by Johnson, Yannakakis, and Papadimitriou [9].

Acknowledgments. We would like to thank Derek Corneil, Mike Fellows, and Carla Savage for helpful discussions, Malcolm Smith for work on the figures, and especially Ken Wong for implementing the algorithms and carefully reading the manuscript.

Note added in proof. As was pointed out by L. Babai (private communication), it is easy to contrive a #P-complete object that can be generated quickly by starting with a #P-complete object and pumping up the number of instances by taking the union with an easily counted and easily generated but more numerous object. For example, given a poset $\mathcal{P}$ on n vertices, consider the set $\{x : x \text{ is an ideal of } \mathcal{P}\} \cup \{x : x \subseteq [n^2]\}$. This set is #P-complete to count and, given an appropriate representation, is easy to generate in constant amortized time.

A counterexample on twenty-four vertices to Conjectures 5.3 and 5.4 has recently been found by J. van den Heuvel (private communication).

REFERENCES

[1] *Webster's New Collegiate Dictionary*, Merriam, Springfield, MA, 1980.

[2] M. Aigner, *Combinatorial Theory*, Springer-Verlag, New York, 1979.

[3] V. Bouchitte and M. Habib, *The calculation of invariants for ordered sets*, Algorithms and Order, I. Rival, ed., Kluwer Academic Publishing, Dordrecht, 1989, pp. 231–279.

[4] G. Brightwell and P. Winkler, *Counting Linear Extensions is #P-complete*, Order, 8 (1992), pp. 225–242.

[5] M. Buck and D. Wiedemann, *Gray codes with restricted density*, Discrete Math., 48 (1984), pp. 163–171.

[6] P. Eades, M. Hickey, and R. C. Read, *Some Hamilton paths and a minimal change algorithm*, J. Assoc. Comput. Math., 31 (1984), pp. 19–29.

[7] H. Fleischner, *The square of every two-connected graph is Hamiltonian*, J. Combin. Theory Ser. B, 16 (1974), pp. 29–34.

[8] B. Jackson and N. C. Wormald, *k-walks of graphs*, Austral. J. Combin., 2 (1990), pp. 135–146.

[9] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, *On generating all maximal independent sets*, Inform. Process. Lett., 27 (1988), pp. 119–123.

[10] S. M. Johnson, *Generation of permutations by adjacent transpositions*, Math. Comp., 17 (1963), pp. 282–285.

[11] A. D. Kalvin and Y. L. Varol, *On the generation of all topological sortings*, J. Algorithms, 4 (1983), pp. 150–162.

[12] D. E. Knuth and J. Szwarcfiter, *A structured program to generate all topological sorting arrangements*, Inform. Process. Lett., 2 (1974), pp. 153–157.

[13] J. K. Lenstra, A. Rinnooy Kan, and P. Brucker, *Complexity of machine scheduling problems*, in Ann. Discrete Math., Vol. 1, North-Holland, Amsterdam and New York, 1977, pp. 343–362.

[14] G. Pruesse, *A generalization of Hamiltonicity*, Tech. report 236/90, University of Toronto, Toronto, Ontario, 1990.

[15] G. Preusse and F. Ruskey, *Generating the linear extensions of certain posets by transpositions*, SIAM J. Discrete Math., 4 (1991), pp. 413–422.

[16] F. Ruskey, *Adjacent interchange generation of combinations*, J. Algorithms, 9 (1988), pp. 162–180.

[17] ———, *Research problem 90*, Discrete Math., 70 (1988), pp. 111–112.

[18] ———, *Generating linear extensions of posets by transpositions*, J. Combin. Theory Ser. B, 54 (1992), pp. 77–101.

[19] M. Sekanina, *On an ordering of the set of vertices of a connected graph*, Publ. Fac. Sci. Brno., 412 (1960), pp. 137–141.

[20] G. Stachowiak, *Hamilton paths in graphs of linear extensions for unions of posets*, SIAM J. Discrete Math., 5 (1992), pp. 199–206.

[21] R. P. Stanley, *Enumerative Combinatorics, Vol.* I, Wadsworth, Belmont, CA. 1986.

[22] H. Steinhaus, *One Hundred Problems in Elementary Mathematics*, Pergamon Press, New York and Oxford, 1963.

[23] H. F. Trotter, *Algorithm 115*: *Perm*, Comm. Assoc. Comput. Math., 5 (1962), pp. 434–435.

[24] Y. L. Varol and D. Rotem, *An algorithm to generate all topological sorting arrangements*, Computer J., 24 (1981), pp. 83–84.

[25] M. B. Wells, *The elements of combinatorial computing*, Pergamon Press, New York and Oxford, 1971.

[26] H. S. Wilf, *Combinatorial Algorithms, An Update*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 387–397, April 1994

RANDOMIZED PARALLEL ALGORITHMS FOR MATROID UNION AND INTERSECTION, WITH APPLICATIONS TO ARBORESENCES AND EDGE-DISJOINT SPANNING TREES*

H. NARAYANAN†, HUZUR SARAN‡, AND VIJAY V. VAZIRANI‡

Abstract. The strong link between matroids and matching is used to extend the ideas that resulted in the design of random NC (RNC) algorithms for matching to obtain RNC algorithms for the matroid union, intersection, and matching problems, and for linearly representable matroids. As a consequence, RNC algorithms for the well-known problems of finding an arboresence and a maximum cardinality set of edge-disjoint spanning trees in a graph are obtained. The key tools used are linear algebra and randomization.

Key words. parallel algorithms, randomized algorithms, graph algorithms, matroids

1. Introduction.

We obtain random NC (RNC) algorithms for the fundamental problems of matroid union and matroid intersection, for linearly representable matroids.

A key step in the matroid union algorithm is a simple method, using randomization, for obtaining a linear representation for the union (it is well known that the union will be a matroid, and will also be linearly representable). Once this is done, it is straightforward to find a maximal independent set in the union matroid. This set is partitioned into independent sets in the corresponding matroids using our RNC matroid intersection algorithm.

An RNC algorithm for the matroid intersection problem, which is a generalization of bipartite matching, is obtained using the Binet-Cauchy Theorem and the Isolating Lemma of [MVV]. Similar techniques, together with a theorem of Lovász [Lo2], yield an RNC algorithm for the matroid matching problem as well. This problem, also called the matroid parity problem, generalizes matroid intersection and general graph matching.

Using the matroid union and intersection algorithms, we obtain RNC algorithms for the matroid covering and packing problems. For the case of a graphic matroid, these correspond to the well-known problems of finding an arboresence in a graph (i.e., the minimum number of forests that cover all edges), and of finding a maximum cardinality set of edge-disjoint spanning trees in a graph. The latter has applications to fault tolerant communications [IR], [Gu]. It also has applications in the analysis of electric networks [IF], [OIW] and in the study of rigidity of structures [LY].

The first polynomial time algorithms for matroid union and matroid intersection were given by Edmonds [Ed1], and Lovász [Lo2] gave the first such algorithm for the matroid matching problem for linearly representable matroids.

Our results draw on the ideas used in the design of RNC algorithms for the maximum matching problem [KUW1], [MVV]. Broadly speaking, the solution for matching consisted of taking the problem into linear algebra, and the use of randomization for extracting a maximum matching. Indeed, we are also forced to work within the realm of linear algebra; our results hold only for linearly representable matroids. However, since almost all useful matroids have this property, this is not a major restriction.

It is interesting to see matroid theory and the matching problem playing their typical roles once again—the former unifies and generalizes concepts, and the latter "serves as an archetypical example of how a 'well-solvable' problem can be studied" (quoted from [LP]).

2. Preliminaries.

Let us first review some basic definitions and notation from matroid theory; details may be found in [Ai], [La], [We].

*Received by the editors February 7, 1991; accepted for publication (in revised form) January 11, 1993.

†Department of Electrical Engineering Indian Institute of Technology, Bombay 400 076, India.

‡Department of Computer Science and Engineering, Indian Institute of Technology, Delhi 110016, India.

A *matroid* M is a finite set $S = \{e_1, \dots, e_n\}$ together with a collection $\mathcal{I}$ of subsets of S (called *independent sets*) such that the following conditions hold.

1. $\emptyset \in \mathcal{I}$.
2. If $X \in \mathcal{I}$ and $Y \subseteq X$ then $Y \in \mathcal{I}$.
3. If $U, V \in \mathcal{I}$ with $|U| = |V| + 1$ then there exists $x \in U - V$ such that $V \cup \{x\} \in \mathcal{I}$.

A subset of S that is not independent is called *dependent*. The *rank* of a set $A \subseteq S$, denoted by $\rho(A)$, is the cardinality of a maximal independent subset of A. The rank of the matroid M is the rank of S. A maximal independent set of M is called a *base*. A fundamental consequence of the matroid axioms is that all bases of a matroid have the same cardinality, thus every maximal independent set is actually maximum.

A matrix B over a field F is a *linear representation* of M if there is a one-to-one correspondence between the elements of S and the columns of B, such that a subset of S is independent in M if and only if the corresponding set of columns of B is independent over F. If the rank of M is r then B need only have r rows. A matroid M is said to be *linearly representable* over F if there exists such a matrix.

For convenience, we will restrict our attention to matroids that are linearly representable over the rationals; it is easy to check that our results are valid as long as $|F|$ is large enough as a function of n and r. Henceforth, "representable" will mean "representable over $\mathbf{Q}$."

Given a graph G, the matroid $\hat{G}$ obtained by taking S to be the set of edges of G and $\mathcal{I}$ to be the forests of G, is called a *graphic* matroid. Every base of $\hat{G}$ corresponds to a spanning forest in G. It is easily seen that if we delete any one row of the vertex-edge incidence matrix of G, we obtain a linear representation of $\hat{G}$ over $GF(2)$. A linear representation of $\hat{G}$ over $\mathbf{Q}$ is obtained by changing one 1 to a -1 in each column of the above matrix.

The *dual* of a matroid M, denoted by M^*, is the matroid whose independent sets are $\{X : \exists B \text{ a base of } M \text{ such that } X \subseteq S - B\}$ i.e., the independent sets of M^* are the subsets of complements of bases in M.

Given a set S and a partition $S_1, S_2, \dots, S_k$ of S, the matroid obtained by taking $\mathcal{I} = \{I \subseteq S : |I \cap S_j| \leq 1, 1 \leq j \leq k\}$ is called a *partition* matroid. It is easily seen that partition matroids are linearly representable (pick k linearly independent vectors and let all elements in S_i be represented by the ith vector).

Let $M_1, M_2, \dots, M_k$ be matroids on S. Let $\mathcal{I} = \{X : X = X_1 \cup \dots \cup X_k$ such that X_i is independent in $M_i, 1 \leq i \leq k\}$. Then $\mathcal{I}$ is the collection of independent sets of a matroid on S. We call this matroid the *union* of $M_1, \dots, M_k$ and denote it by $\bigvee_{i=1}^k M_i$. If X is an independent set in $\bigvee_{i=1}^k M_i$, then a partition of X into sets $X_1, \dots, X_k$ such that X_i is independent in M_i is called a *proper* partition of X.

The intersection of matroids can be defined analogously. However, the collection of sets so obtained does not form the independent sets of a matroid. As a consequence, a maximal set in this collection need not be maximum. The problem of finding a maximum cardinality set in the intersection of two matroids is called the *matroid intersection* problem.

The *matroid matching problem* is: given matroid $M = (S, \mathcal{I})$, and a partition of S into pairs $(x_1, x_2), \dots, (x_{2n-1}, x_{2n})$, pick the largest number of pairs so that the picked elements form an independent set.

The (i, j)th entry of a $n \times n$ matrix A will be denoted by $A(i, j)$ and its determinant will be denoted by $|A|$:

$$|A| = \sum_{\sigma} \text{sign}(\sigma)\text{value}(\sigma)$$

Here, the sum is taken over all permutations of $\{1, \dots, n\}$, sign(σ) is $+1$ if σ is even and -1 if σ is odd, and value$(\sigma) = \prod_{i=1}^n A(i, \sigma(i))$.

In this paper, we state all our results assuming the arithmetic CREW PRAM model of computation [KR]. The arithmetic operations we require are addition, multiplication, and subtraction of $O(n)$ bit numbers, where n is the size of the ground set. We use $M(n)$ to denote the number of processors required to multiply two $n \times n$ matrices in $O(\log n)$ time, $M(n) = o(n^{2.376})$ [CW].

3. Obtaining a representation for the union matroid. If matroids $M_1, M_2, \ldots, M_k$ over set S are linearly representable, then the union $\bigvee_{i=1}^{k} M_i$ is also linearly representable. However, the standard way of constructing such a representation does not seem to be parallelizable. We first give an *RNC* algorithm for constructing a linear representation of the union of k matroids, for any $k \geq 1$.

We will give details for $k = 2$; the generalization will be obvious. Let the ranks of M_1 and M_2 be r_1 and r_2, respectively. Let A_1 be an $r_1 \times n$ matrix representing M_1 and A_2 be an $r_2 \times n$ matrix representing M_2. Obtain A'_2 from A_2 by multiplying the columns of A_2 by distinct variables $x_1, x_2, \ldots, x_n$, i.e., $A'_2(i, j) = A_2(i, j) \cdot x_j$. It is easy to verify that a set of columns of A'_2 is independent if and only if the corresponding set of columns of A_2 is independent. (Linear independence for matrices containing variables is defined in a natural way, using operations over the field of multivariate polynomials.)

Define B, an $(r_1 + r_2) \times n$ matrix, whose first r_1 rows are the same as the rows of A_1 and the last r_2 rows are the same as the rows of A'_2, i.e.,

$$B(i, j) = \begin{cases} A_1(i, j) & \text{if } i \leq r_1, \\ A'_2(i - r_1, j) & \text{otherwise.} \end{cases}$$

LEMMA 3.1. *A set $S' \subseteq S$ is independent in $M_1 \vee M_2$ if and only if the corresponding columns of B are linearly independent.*

Proof. Let us first state a key property of matrix B that makes the lemma hold. Let C be a $(k + l) \times (k + l)$ submatrix of B that picks k rows from A_1 and l rows from A'_2, and some set of $k + l$ columns. Corresponding to a set of l of these columns, we define a $k \times k$ submatrix C_1 and an $l \times l$ submatrix C_2 of C as follows: C_2 consists of l rows of A'_2 and these l columns, and C_1 consists of the unpicked rows and columns of C. Let the indices of the l columns of C_2 be $i_1, \ldots, i_l$ in B. Consider $|C| = \sum_{\pi} \text{sign}(\pi)\text{value}(\pi)$, where as usual, the sum runs over all permutations π mapping rows of C onto columns of C. Let P be the set of permutations mapping rows of C_1 onto columns of C_1, and rows of C_2 onto columns of C_2. Then it is easy to check that

$$\sum_{\pi \in P} \text{sign}(\pi)\text{value}(\pi) = |C_1||C_2|;$$

this term is of the form $cx_{i_1}, \ldots, x_{i_l}, c \in \mathbf{Q}$. Furthermore, a monomial of this form is not generated by any permutation $\pi \notin P$. Therefore, if C_1 and C_2 are both non-singular then so is C. Similarly, if C is non-singular then for a suitable choice of $i_1, \ldots, i_l$ (corresponding to a monomial with non-zero coefficient in $|C|$), C_1 and C_2 must be non-singular.

The rest of the proof is now straightforward. Suppose $S' \subseteq S$ is independent in $M_1 \vee M_2$. Let (S_1, S_2) be a proper partition of S', and let $|S_1| = k$ and $|S_2| = l$. Then the columns of $A_1(A'_2)$ corresponding to $S_1(S_2)$ are linearly independent, and one can pick $k(l)$ rows to get a $k \times k(l \times l)$ non-singular matrix $C_1(C_2)$. Then by the above stated property, the $(k+l) \times (k+l)$ matrix C consisting of all the picked rows and columns is non-singular. Hence the columns of B corresponding to the elements of S' are linearly independent.

To prove the other direction, pick a maximal non-singular square submatrix, C, of the linearly independent columns of B. The columns in C correspond to some set $S' \subseteq S$. By the above-stated property, there exist non-singular submatrices C_1 and C_2 of C. These yield a partition of S' into (S_1, S_2) such that S_i is independent in M_i, $i = 1, 2$. This implies that S' is independent in $M_1 \vee M_2$. □

The construction given above clearly generalizes to k matroids $M_1, \dots, M_k$ of rank $r_1, \dots, r_k$: multiply the columns of $M_2, \dots, M_k$ with distinct variables, $x_1, \dots, x_{(k-1)n}$, and "stack up" the resulting matrices, together with M_1, to obtain an $(r_1 + \cdots + r_k) \times n$ matrix B. Since matroid union is associative, the proof that this construction works follows immediately using induction.

LEMMA 3.2. *Given matroids $M_1, \dots, M_k$, the matrix B obtained above is a linear representation of $\bigvee_{i=1}^{k} M_i$.*

We next show, using randomization, how to obtain a linear representation of $\bigvee_{i=1}^{k} M_i$ using only rational entries, i.e., with no variables.

LEMMA 3.3. *Let $\tilde{B}$ be obtained from B by substituting each indeterminate randomly and independently from $\{1, \dots, 2n\binom{n}{n/2}\}$. Then*

$$Pr\left[\tilde{B} \text{ is a linear representation of } \bigvee_{i=1}^{k} M_i\right] > \frac{1}{2}$$

Proof. Note that it is enough to show that our substitution maintains the independence of every base of B. Let the rank of the union matroid be r. Then, it can have at most $\binom{n}{r}$ distinct bases. Corresponding to each base β there is an $r \times r$ non-singular submatrix, C_β, of B. Consider the polynomial

$$p = \Pi_\beta |C_\beta|,$$

where the product is taken over all bases of B. The degree of p is no more than $\binom{n}{r} r$. Now it is enough to obtain a substitution for the indeterminates such that p evaluates to a non-zero value. By Lemma 3.4, it is sufficient to substitute for the variables randomly and independently from the integers $\{1, 2, \dots, 2r\binom{n}{r}\}$, i.e., using $O(n)$ bits for each number, since $2r\binom{n}{r} \leq 2n2^n$. □

LEMMA 3.4. [Sc], [Zi] *Let $p(x_1, x_2, \dots, x_n)$ be a non-trivial polynomial of degree d over field F and let $S \subseteq F$. If $x_1, x_2, \dots, x_n$ are randomly and independently chosen from S then,*

$$Pr[\textit{the substitution is a zero of } p] \leq d/|S|.$$

Hence we get the following theorem.

THEOREM 3.5. *Given the linear representations of k matroids $M_1, M_2, \dots, M_k$ with ranks $r_1, r_2, \dots, r_k$, there is an RNC^0 algorithm for obtaining a representation of the union using $n(\sum_{i=1}^{k} r_i) \leq n^2 k$ processors.*

Once we have a representation, $\tilde{B}$, for the union matroid $M = \bigvee_{i=1}^{k} M_i$, we can find the lexicographically first base, I, for M by finding the lexicographically first base of $\tilde{B}$. The lexicographically first base of a matrix can be obtained in NC^2 by computing in parallel the rank of first i columns, for $i = 1, \dots, n$, and choosing indices where rank increases. We get the following corollary.

COROLLARY 3.6. *Given k matroids $M_1, \dots, M_k$ via their linear representations over $\mathbf{Q}$, there is an RNC^2 algorithm that uses $O(\max\{kn^2, nM(n)\})$ processors for obtaining the lexicographically first maximal independent set in the union $\bigvee_{i=1}^{k} M_i$.*

Remark. Notice that even if the random substitution was a zero of p, if a set of columns is independent in $\tilde{B}$ then it must be independent in B. Therefore I is guaranteed to be independent in M.

Next we would like to obtain a proper partition of I. For this we need a parallel algorithm for the matroid intersection problem.

4. Matroid intersection and matroid matching.

4.1. Matroid intersection. We use the Isolating Lemma of [MVV] and the Binet-Cauchy Theorem to obtain an RNC algorithm to compute the maximum cardinality set in the intersection of two matroids. Recall that the Binet-Cauchy Theorem states that, given two $n \times m$ matrices A and B,

$$|AB^T| = \sum_{\alpha} |A_\alpha||B_\alpha|$$

where the sum is taken over all possible ways, α, of choosing n columns out of m, and by $A_\alpha(B_\alpha)$ we mean the $n \times n$ submatrix of $A(B)$ consisting of columns chosen by α.

We shall need the following slightly modified version of the Isolating Lemma (this is a straightforward extension of the original lemma).

LEMMA 4.1 (Isolating Lemma [MVV]). *Let $(X, \mathcal{F})$ be a set system, where $X = \{x_1, x_2, \ldots, x_n\}$ and $\mathcal{F} \subseteq 2^X$ is a family of subsets of X. Let $\{c_1, c_2, \ldots, c_n\}$, $c_i \in \mathbf{Q}$ be a set of initial weights on the x_i's. Pick $w_1, w_2, \ldots, w_n$ randomly and independently from $[1, \ldots, 2n]$. Define the weight of x_i to be $c_i + w_i$, and let the weight of a set be the sum of the weights of its elements. Then,*

$$Pr[\exists \textit{ unique minimum weight set in } \mathcal{F}] \geq \frac{1}{2}.$$

THEOREM 4.2. *There is an RNC^2 algorithm using $O(n^{4.5})$ processors for the matroid intersection problem for linearly representable matroids.*

Proof. Let the two matroids M_1 and M_2 be represented by the matrices A and B, respectively. Without loss of generality assume that A and B have the same number of rows, say r (since the smaller matrix can be padded with rows of zeroes). In general, the required maximum cardinality subset of S may have fewer than r elements; if so, by the Binet-Cauchy Theorem $|AB^T| = 0$, and we get no information from this determinant. We shall overcome this problem by first augmenting A and B with extra columns. Let $e_1, e_2, \ldots, e_r$ be the unit vectors over $\mathbf{Q}^r$. Obtain $r \times r^2$ matrices C and D such that for each $1 \leq i, j \leq r$, there is an index $1 \leq k \leq r^2$, such that the kth column of C is e_i and the kth column of D is e_j.

Let $[A, C]$ represent the $r \times (n + r^2)$ matrix whose first n columns are the same as those of A, and the last r^2 columns are the same as those of C. Similarly, obtain $[B, D]$. The augmented matrices have the following property: any subset of indices from $[1, \ldots, n]$ such that the corresponding columns are linearly independent both in A and in B can be extended to a set of r indices from $[1, \ldots, n + r^2]$ such that the corresponding columns are linearly independent both in $[A, C]$ and $[B, D]$.

Randomly and independently pick $w_1, \ldots, w_{n+r^2}$ from $[1, \ldots, 2(n+r^2)]$. For $1 \leq i \leq n$, multiply the ith column of $[A, C]$ by x^{w_i}, and for $n + 1 \leq i \leq r^2 + n$, multiply the ith column of $[A, C]$ by $x^{w_i + 2r(n+r^2)}$ to obtain a new $r \times (n + r^2)$ matrix E. Compute $|E[B, D]^T|$. This can be done in NC^2 using $O(n^{4.5})$ processors (see [BCP]).

We now apply the Isolating Lemma. Here the set system, X, consists of the set of $n + r^2$ indices and $\mathcal{F}$ consists of all choices of r indices such that the corresponding columns are independent both in $[A, C]$ and in $[B, D]$. Then with probability at least 1/2, the minimum weight set of indices, α, is unique for the chosen weight distribution, and has weight w_α. By the Binet-Cauchy Theorem $|E[B, D]^T| \not\equiv 0$, since the coefficient of the minimum power of x (i.e., x^{w_α}) is non-zero. The minimum weight set of indices α can be obtained as follows:

```
For $i=1$ to $n+r^2$, in parallel do:
    decrease $w_i$ by 1, keeping the
    rest of the weights unchanged,
    and compute $|E[B,D]^T|$. If
    the coefficient of $x^{w_\alpha}$ changes,
    then pick index $i$.
end;
```

The weight of any column of C is greater than the sum of weights of all columns of A, irrespective of the random choices made. Therefore, α must pick as many indices from $[1,\dots,n]$ as possible. Hence, the subset of α in the range $[1,\dots,n]$ will constitute a maximum cardinality subset of S that is independent in both matroids. □

Remark. The algorithm given above extends to the weighted matroid intersection problem, provided the weights are given in unary. Let the ith element of the ground set have weight W_i; find a minimum weight maximum cardinality set in the intersection of the two matroids. In this case, we multiply the ith column of $[A,C]$ by $x^{w_i+2(n+r^2)W_i}$ for $1 \le i \le n$, and by $x^{w_i+2r(n+r^2)W}$ for $n < i \le n+r^2$, where W is the weight of the heaviest ground set element. Extending this to the case of binary weights is left open.

4.2. Matroid matching. In this section we shall give an RNC algorithm for the *matroid matching problem* for linearly representable matroids: given m pairs of vectors over $\mathbf{Q}^{2n}$, $\{a^{(1)},b^{(1)}\},\dots,\{a^{(m)},b^{(m)}\}$, pick the largest number of pairs so that the picked vectors are linearly independent. Let us first define the *wedge product* of vectors a, b over $\mathbf{Q}^{2n}$ as an $2n \times 2n$ matrix A such that

$$A(i,j) = a_ib_j - a_jb_i.$$

Denote this by $(a \wedge b)$. We will need the following theorem of Lovász.

THEOREM 4.3. [Lo2], [LP]. *Let* $\{a^{(1)},b^{(1)}\},\dots,\{a^{(m)},b^{(m)}\}$ *be pairs of vectors over* $\mathbf{Q}^{2n}$. *Then there exists a set of n pairs whose union is a basis iff* $|B| \not\equiv 0$, *where*

$$B = \sum_{i=1}^{m}(a^{(i)} \wedge b^{(i)})x_i.$$

Here the x_i's are distinct indeterminates.

THEOREM 4.4. *There is an* RNC^2 *algorithm using* $O(n^{4.5})$ *processors for the matroid matching problem for linearly representable matroids.*

Proof. As in Theorem 4.1, we will first deal with the issue that there may not be n pairs (a_i,b_i) whose union is a basis, by throwing in $\binom{2n}{2}$ additional pairs of vectors. Let $e_1,\dots,e_{2n}$ be the unit vectors in $\mathbf{Q}^{2n}$. The added vectors are chosen in such a way that for every pair of indices (i,j), $1 \le i < j \le 2n$, (e_i,e_j) is included.

As before, as a result of the augmentation, any subset of the original m pairs that are linearly independent can be extended to n pairs using the added pairs. Let

$$B = \sum_{i=1}^{m+\binom{2n}{2}} (a^{(i)} \wedge b^{(i)})x_i.$$

Randomly and independently pick $w_1,\dots,w_{m+\binom{2n}{2}}$ from $[1,\dots,2(m+\binom{2n}{2}))]$. For $1 \le i \le m$, substitute $x_i = x^{w_i}$, and for $m+1 \le i \le m+\binom{2n}{2}$, substitute $x_i = x^{w_i+2n(m+\binom{2n}{2}))}$, and compute $|B|$.

Note that B is skew-symmetric, and hence $|B| = (pf(B))^2$, where $pf(B)$ is the pfaffian of B. Lovàsz [Lo2] shows that $pf(B)$ is linear in each of the variables x_i and each monomial in the polynomial $pf(B)$ is the product of n distinct variables x_i. We will now apply the Isolating Lemma with X as the set of $m + \binom{2n}{2}$ indices, and $\mathcal{F}$ consisting of all choices of n indices such that the corresponding monomial has non-zero coefficient in $pf(B)$. Then, after the random substitution given above, with probability at least $\frac{1}{2}$, there is a unique term having the minimum power of x (say x^w) in $pf(B)$. Consequently, the polynomial $|B|$ will also contain a unique term having minimum power of x, x^{2w}. Hence $|B| \not\equiv 0$. The indices contributing to this term can be obtained as in Theorem 4.1, by decreasing w_i by 1 in parallel for each i. By the choice of weights, the number of indices in the range $[1, \ldots, m]$ is maximized. Hence, this set of indices is a solution to the matroid matching problem. □

As in the case of matroid intersection, the above algorithm extends to the weighted problem, if the weights are given in unary. Once again, the weighted matroid matching problem for binary weights is left open.

5. Partitioning the independent set. Suppose set I is independent in $M_1 \vee M_2$. In this section, we show how to obtain a proper partition of I.

5.1. Obtaining a representation for the dual matroid. First, we need a parallel algorithm for obtaining a representation of the dual matroid. The standard method parallelizes in a straightforward manner; essentially, it involves finding a base for the null-space of A, where A is a $r \times n$ matrix representing M. Denote the submatrix of A consisting of the first r (last $n-r$) columns of A by $A_1(A_2)$. The columns of A can be permuted in NC to ensure that A_1 is non-singular.

```
For each $n-r$ dimensional unit vector $e_i$,
$1 \leq i \leq n-r$ in parallel do:
  Compute $x_i = -A_1^{-1}A_2e_i$.
  Output the $n$ dimensional vector
    whose first $r$ components
    consist of $x_i$ and the remaining
    $n-r$ components consist of $e_i$.
end;
```

Let C be the matrix consisting of these $n-r$ n-dimensional column vectors (in any order). Then $B = C^T$ is a linear representation for M^* [We]. Since matrix inversion is in NC^2 (see [Cs]) we get the following lemma.

LEMMA 5.1. *There is an NC^2 algorithm using $O(nM(n))$ processors for obtaining the linear representation of the dual, M^*, of a linearly representable matroid M.*

5.2. The partitioning algorithm. We may assume without loss of generality that I is the ground set for M_1 and M_2 (if not, we can restrict M_1 and M_2 to I, and pick only the corresponding columns of matrices A and B). The partitioning algorithm is as follows:

```
1. Find a representation $B^*$ for $M_2^*$.
2. Apply the matroid intersection algorithm on $A$ and $B^*$.
3. Let $I_1$ be the set found in step 2.  Output $(I_1, I_2)$ where
   $I_2 = I - I_1$.
```

LEMMA 5.2. *Assuming that the matroid intersection algorithm in step* 2 *is successful, (I_1, I_2) is a proper partition of I.*

Proof. It is sufficient to prove that I_2 is independent in M_2, since clearly I_1 is independent in M_1. Since I_1 is independent in M_2^*, I_2 contains a base of M_2. We will finish the proof by showing that $|I_2|$ must equal the cardinality of a base of M_2.

Let (I_1', I_2') be a proper partition of I that maximizes $|I_2'|$. Then I_2' must be a base of M_2; if not, we should be able to move an element from I_1' to I_2' (note that $I_1' \cup I_2'$ is the ground set of M_2). Clearly, I_1' is independent in M_1 and M_2^*. Therefore $|I_1| \geq |I_1'|$, and so $|I_2| \leq |I_2'|$. Since I_2 contains a base of M_2, we have that $|I_2| = |I_2'|$. The lemma follows. □

When we are given I over $M_1 \vee M_2 \vee \cdots \vee M_k$ using the previous algorithm, we can first partition it into I_1 and I_2 such that I_1 is independent in $M_1 \vee \cdots \vee M_{k/2}$ and I_2 is independent in $M_{k/2+1} \vee \cdots \vee M_k$. For this, we need to construct representations for the two matroids $M_1 \vee \cdots \vee M_{k/2}$ and $M_{k/2+1} \vee \cdots \vee M_k$. Clearly, this just involves choosing the appropriate rows of the representation of $M_1 \vee M_2 \vee \cdots \vee M_k$. Now recursively, in parallel, solve the problem for I_1 and I_2. Note that this will take at most $\log n$ iterations of the above algorithm.

THEOREM 5.3. *There is an RNC^3 algorithm using $O(n^{4.5})$ processors for the following problem. Given matroids $M_1, \ldots, M_k$, via their linear representations and an independent set I in $\bigvee_{i=1}^k M_i$, find a proper partition of I.*

6. Covering and packing problems. We give RNC algorithms for solving the covering and packing problems for a linearly representable matroid M. Let M^k denote $\bigvee_{i=1}^k M$.

The *matroid covering problem* is: Find a minimum cardinality collection $\mathcal{C}$ of independent sets in M such that $\bigcup_{X \in \mathcal{C}} X = S$. Let $|\mathcal{C}| = k$, then $k \leq n$, assuming that there are no trivial elements in S (i.e., elements that participate in no independent set). Clearly $k = \min\{i : \rho(M^i) = n\}$. Thus, we can carry out a binary search in the interval $[1, \ldots, n]$ to obtain k. This would involve at most $\log n$ iterations of the matroid union algorithm. We can then partition S into k independent sets using the algorithm of Theorem 5.1 to get the required cover.

THEOREM 6.1. *There is an RNC^3 algorithm using $O(n^{4.5})$ processors for obtaining a minimum cardinality cover by independent sets of a linearly representable matroid M.*

Since the graphic matroid of a graph is linearly representable (see §2), we get the following corollary.

COROLLARY 6.2. *There is an RNC^3 algorithm using $O(n^{4.5})$ processors for obtaining an arboresence of a graph G.*

The *matroid packing problem* is: Find a maximum cardinality set of mutually disjoint bases of M. Let k be the cardinality of this set and r be the rank of M. Clearly, $k = \max\{i : \rho(M^i) = ri\}$, and $k \leq \lfloor n/r \rfloor$. To find k, we can do a binary search over the range $1, \ldots, \lfloor n/r \rfloor$. A partition of any base of M^k will then give the required set of bases.

THEOREM 6.3. *There is an RNC^3 algorithm using $O(n^{4.5})$ processors for obtaining a maximum cardinality set of disjoint bases of a linearly representable matroid M.*

COROLLARY 6.4. *There is an RNC^3 algorithm using $O(n^{4.5})$ processors for finding a maximum cardinality set of edge-disjoint spanning trees of a graph G.*

7. Las Vegas extensions. The algorithms presented so far have all been Monte Carlo, i.e., they work with probability at least $\frac{1}{2}$. We now give parallel algorithms for verifying the solutions obtained, thereby giving their Las Vegas extensions, i.e., the running time of the algorithm is probabilistic; however, it is guaranteed to produce the correct solution.

The matroid intersection algorithm can be made Las Vegas as follows. Suppose the intersection computed is I. Now for each element $e \in S - I$, check using Edmonds' matroid intersection algorithm whether the set can be augmented with e (this will be a transitive closure computation in an appropriately defined graph, see [PS]). If this fails for each element, then I is the largest set in this intersection.

Next, we make the matroid union algorithm Las Vegas. Let I be the lexicographically first maximal independent set found in $M_1 \vee \cdots \vee M_k$, and Let $I_1, \ldots, I_k$ be the partition of I obtained. Notice that randomization is used at two points: in finding I, and in partitioning it. Because of Remark 3.1, I must be independent, though it may not be the lexicographically first maximal independent set. We can verify this solution as follows:

1. Verify that I_i is independent in M_i, $1 \leq i \leq k$; if not, halt. This can be done easily since linear representations of M_i are available. (Notice that a failure here indicates a bad choice in the randomization done in the partitioning phase. Hence, if this test is passed, then the partitioning of I is correct.)

2. Verify that I is the lexicographically first maximal independent set of $M_1 \vee \cdots \vee M_k$ as follows: Let $e_1, \ldots, e_n$ be the ordering on the elements of S, and let $I^{(i)}$ denote the restriction of I to $\{e_1, \ldots, e_i\}$. It is sufficient to verify that for each i, $1 \leq i < n$: if $e_{i+1} \notin I$ then $I^{(i)} \cup \{e_{i+1}\}$ is dependent in the union matroid. This can be done in parallel for each i, as in the matroid intersection case given above. Notice that a partition of $I^{(i)}$ is available.

We next turn to covering. Suppose our algorithm found that $k = \min\{i : S \text{ is independent in } M^i\}$. Now we only need to verify that S is not independent in M^{k-1}. This can easily be done by finding a maximal independent set in M^{k-1}. Finally, the quantity k returned by the packing procedure can be verified by checking if $\rho(M^{k+1}) < r(k+1)$, where r is the rank of M. We leave the open problem of obtaining a Las Vegas extension for the matroid matching problem.

8. Discussion and open problems. (1) Our results hold only for linearly representable matroids. Although almost all interesting matroids have this property, it is still interesting to check whether there exist fast parallel algorithms for the matroid union and intersection problems when the matroids are not linearly representable. We may assume that a rank oracle or an independence oracle is given for the matroids. In the sequential setting, either of these oracles suffices for running Edmonds' algorithms. The importance of linear representability to algorithm design has been noted previously. For example, Lovász [Lo1] has given a polynomial time algorithm for the matroid matching problem if the given matroid has a linear representation; however, the general problem is not polynomial time solvable [Lo1], [JK] (interestingly, this result does not depend on the $P \neq NP$ hypothesis). In the spirit of [Lo1], [JK] can one prove negative results in the parallel setting? For other work along these lines see [KUW2].

(2) The ideas in this paper can also be applied to finding branching(s) in digraphs. In a directed graph $G = (V, E)$, a *branching rooted at vertex* v is an acyclic subgraph of G in which v has indegree zero, and every other vertex has indegree one.

Lovász [Lo3] gives an NC^2 algorithm for finding a branching in a directed graph. It is well-known that sequentially a branching can be found using matroid intersection. Let us first remark that our parallel matroid intersection algorithm gives an RNC^2 algorithm for finding a branching, although the algorithm in [Lo3] is superior not only because it is deterministic, but also because it uses fewer processors. The advantage of the matroid approach is that it extends to the problem of obtaining k edge-disjoint branchings rooted at v, using the following theorem.

THEOREM 8.1. [Ed2] *A set $E' \subseteq E$ can be partitioned into k edge-disjoint branchings rooted at v if and only if*

(i) *when considered as a set of undirected edges, E' can be partitioned into k spanning trees,*

(ii) *every vertex other than v has indegree k.*

Let M be the graphic matroid on E obtained by ignoring edge directions, and let P be the partition matroid on E where the elements of the ith partition are the edges that point into the vertex i. To find a set E' (if it exists) of edges that can be partitioned into k edge-disjoint branchings rooted at v, obtain a maximum cardinality set in the intersection of M^k and P^k. The problems of partitioning E' into k branchings, and of making this algorithm Las Vegas are left as open problems.

(3) What is the parallel complexity of finding

(a) a lexicographically first intersection of two matroids

(b) a lexicographically first maximum cardinality intersection of two matroids.

Special cases of (a) and (b) are lexicographically first maximal and maximum matching, respectively, in bipartite graphs. Analogous problems can also be stated for the graphic matroid. Finding a lexicographically first maximal matching is known to be CC-complete. On the other hand, the parallel complexity of lexicographically first maximum matching is unresolved. Is problem (a) CC-complete?

Acknowledgments. We wish to thank Prasoon Tiwari and Umesh Vazirani for valuable discussions during which a restricted version of Theorem 4.1 was obtained. Also we would like to thank Naveen Garg, Samir Khuller, Shachin Maheshwari, Sachin Patkar, Milind Sohoni, and Prakash Sunderasan for useful discussions.

REFERENCES

[Ai] M. Aigner, *Combinatorial Theory*, Springer-Verlag, New York, 1979.

[BCP] A. Borodin, S. A. Cook, and N. Pippinger, *Parallel computation for well-endowed rings and space bounded probabilistic machines*, Inform. and Control 58 (1983), pp. 113–136.

[Cs] L. Csanky, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618–623.

[CW] D. Coppersmith and S. Winograd, *Matrix multiplication via arithematic progressions*, Proc. 19th Annual Symp. on Theory of Computing, 1987, pp. 1–6.

[Ed1] J. Edmonds, *Minimum partition of a matroid into independent subsets*, J. Res. National Bureau of Standards, 69B (1965), pp. 67–72.

[Ed2] ———, *Edge-disjoint branchings*, Combinatorial Algorithms, R. Rustin, ed., Algorithmics Press, New York, 1972, pp. 91–96.

[Gu] D. Gusfield, *Connectivity and edge-disjoint spanning trees*, Inform. Process. Lett., 16 (1983), pp. 87–89.

[IF] M. Iri and S. Fujishige, *Use of matroid theory in operations research, circuits, and systems theory*, Internat. J. Systems Sci., 12 (1981), pp. 27–54.

[IR] A. Itai and M. Rodeh, *The multi-tree approach to reliability in distributed networks*, Proc. 25th Annual Symp. on Foundations of Comp. Sci., 1984, pp. 137–147.

[JK] P. M. Jensen and B. Korte, *Complexity of matroid property algorithms*, SIAM J. Comput., 11 (1983), pp. 184–190.

[KK] G. Kishi and Y. Kajitani, *Maximally distant trees and principal partition of a linear graph*, IEEE Trans. Circuit Theory, CT-16, 3 (1969), pp. 323–330.

[KR] R. M. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science, vol. A, MIT Press/Elsevier, New York, 1990, pp. 869–941.

[KUW1] R. M. Karp, E. Upfal, and A. Wigderson, *Constructing a maximum matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[KUW2] ———, *The complexity of parallel search*, J. Comput. System Sci., 36 (1988), pp. 225–253.

[La] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rhinehart and Winston, New York, 1976.

[Lo1] L. Lovász, *The matroid matching problem*, Algebraic Methods in Graph Theory, II, L. Lovász and V. T. Sós, eds., Colloq. Math. Soc. János Bolyai, vol. 25, Elsevier-North-Holland, Amsterdam, 1981, pp. 495–517.

[Lo2] ———, *On determinants, matchings, and random algorithms*, Fundamentals of Computation Theory, FCT'79, L. Budach, ed., Math Research 2, Akademie-Verlag, Berlin, 1979, pp. 565–574.

[Lo3] ———, *Computing ears and branchings in parallel*, Proc. 26th Annual Symp. on Foundations of Computer Science, 1985, pp. 464–467.

[LP] L. Lovász and M. Plummer, *Matching Theory*, Academic Press, Budapest, 1986.

[LY] L. Lovász and Y. Yemini, *On generic rigidity in the plane*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 91–98.

[MS] E. W. Mayr and A. Subramanian, *The complexity of circuit value and network stability*, submitted to J. Comput. System Sci.

[MVV] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–114.

[Nw] C. St. J. A. Nash-Williams, *Edge-disjoint spanning trees of finite graphs*, J. London Math. Soc., 36 (1961), pp. 445–450.

[PS] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms & Computing*, Prentice Hall, Englewood Cliffs, NJ, 1982.

[OIW] T. Ohtsuki, Y. Ishizaki, and H. Watanabe, *Topological degrees of freedom and mixed analysis of electrical networks*, IEEE Trans. Circuit Theory, CT-17, 4 (1970), pp. 491–499.

[Sc] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, JACM, 27 (1980), pp. 281–292.

[We] D. J. A. Welsh, *Matroid theory*, Academic Press, New York, 1976.

[Zi] R. E. Zippel, *Probabilistic algorithms for sparse polynomials*, Proc. EUROSAM 79, Ng, ed., Springer Lecture Notes in Computer Science, vol. 72 (1979), pp. 216–226.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 398–414, April 1994

ON THE CONSECUTIVE-RETRIEVAL PROBLEM*

R. SWAMINATHAN† AND DONALD K. WAGNER‡

Abstract. A {0, 1}-matrix M has the *consecutive-retrieval property* if there exists a tree T such that the vertices of T are indexed on the rows of M and the columns of M are the incidence vectors of the vertex sets of paths of T. If such a T exists, then T is a *realization* for M. In this paper, an $O(r^2c)$ algorithm is presented to determine whether a given standard, $r \times c$ matrix has the consecutive-retrieval property and, if so, to construct a realization.

Key words. consecutive-retrieval property, tree realization, matrix decomposition, polynomial-time algorithm

AMS subject classifications. 05C05, 68P20

1. Introduction. A {0, 1}-matrix M has the *consecutive-retrieval property* if there exists a tree T such that the vertices of T are indexed on the rows of M and the columns of M are the incidence vectors of the vertex sets of paths of T. (Note that not every path of T needs to correspond to a column of M.) If such a T exists, then T is a *realization* for M and M is *consecutive-retrieval graphic* (abbreviated *cr-graphic*). The *consecutive-retrieval problem* is to determine whether a given {0, 1}-matrix M is cr-graphic and, if so, to construct a realization. This paper presents an $O(r^2c)$ algorithm for the consecutive-retrieval problem, where the given matrix is a standard, $r \times c$ {0, 1}-matrix.

The consecutive-retrieval problem defined above was introduced by Tanaka [20] as a generalization of versions of the problem defined by Ghosh [9] and Lipski [15].

The problem defined by Ghosh is as follows. Let R be a set of *records*, and let $\mathcal{Q}$ be a collection of subsets of R called *queries*. The problem is to determine whether the members of R can be arranged linearly in storage locations in such a way that the members of each $Q \in \mathcal{Q}$ are stored in consecutive locations. One advantage of such a storage arrangement is that any query can be retrieved from storage using just two parameters, namely, a pointer to the beginning of the query and the number of records in the query. Ghosh's problem can be rephrased in the terminology of the present paper as follows. Let M be a matrix, the columns of which are the incidence vectors of the members of $\mathcal{Q}$. Then the problem is to determine whether there exists a path P such that the vertices of P are indexed on the rows of M and the columns of M are the incidence vectors of the vertex sets of subpaths of P. Thus, the consecutive-retrieval problem defined above is an extension of Ghosh's problem from paths to trees. Different aspects of Ghosh's problem have been studied by Ghosh [10], [11], [12], Gupta [13], Kou [14], Patrinos and Hakimi [18], and Waksman and Green [25].

A different rephrasing of Ghosh's problem is that of determining whether the rows of M can be permuted such that the ones in each column occur consecutively. A matrix admitting such a permutation is called a *consecutive-ones matrix*. Such matrices were introduced by Fulkerson and Gross [7], [8] and studied further in Tucker [22], [23], Nakano [16], [17], and Booth and Lueker [4]. In particular, Booth and Lueker gave a linear-time algorithm (linear in the number of nonzeros) for determining whether a given matrix is a consecutive-ones matrix.

Lipski [15] generalized Ghosh's problem as follows. For a given {0, 1}-matrix M, Lipski's problem is that of determining whether there exists an arborescence T such that the vertices of T are indexed on the rows of M and each column of M is the incidence vector of the

*Received by the editors July 16, 1992; accepted for publication (in revised form) February 5, 1993.

†Department of Computer Science, University of Cincinnati, Cincinnati, Ohio 45221 (swamy@jupiter.csm.uc.edu).

‡Mathematical Sciences Division, Office of Naval Research, Arlington, Virginia 22217 (dwagner@onr-hq.navy.mil). The research of this author was partially supported by Office of Naval Research Grant N00014-86-K-0689 at Purdue University.

vertex set of a directed path of T. Thus, the consecutive-retrieval problem of this paper is an undirected version of Lipski's problem. As in Ghosh's problem, if M represents a set of queries, then each query can be retrieved from the arborescence T using just two parameters, the end of the corresponding directed path and its length. Truszczyński [21] gave an $O(c^3 + c^2r)$ algorithm for determining whether M admits such an arborescence. Dietz, Furst, and Hopcroft [5] improved this to linear time. An almost-linear-time algorithm and several different applications of the problem are provided by the authors in [19].

Observe that if a cr-graphic matrix represents a set of queries, then each query can be retrieved from a realization T using just four parameters as follows. First, assign directions to the edges of T so as to make it an arborescence. Then any path of T is either a directed path of the arborescence or is the union of two directed paths. Since each of these directed paths can be retrieved using just two parameters, any path of T corresponding to a query can be retrieved with just four.

An edge analogue of the consecutive-retrieval problem has previously been studied. Define a $\{0, 1\}$-matrix M to be *edge-tree graphic* if there exists a tree T such that the columns of M are the incidence vectors of the edge sets of paths of T. If such a T exists, then T is an *edge-tree realization* for M. The *edge-tree realization problem* is that of determining whether a given matrix M is edge-tree graphic and, if so, to construct an edge-tree realization. A polynomial-time algorithm for this problem was first devised by Tutte [24]. Several other algorithms have been developed, the most efficient of which are the almost-linear-time algorithms of Bixby and Wagner [2] and Fujishige [6].

The algorithm presented here for the consecutive-retrieval problem is reminiscent of Tutte's algorithm [24] for the edge-tree realization problem. The basic idea of the present algorithm is as follows. The algorithm consists of two phases, a decomposition phase followed by a composition phase. In the decomposition phase, the matrix M is recursively decomposed into smaller matrices with the final set of matrices each having exactly one row. Such matrices are obviously cr-graphic. The composition phase starts with realizations of these one-rowed matrices and undoes the recursion of the first phase in an attempt to construct a realization of M.

The remainder of the paper is outlined as follows. Section 2 develops the decomposition phase of the algorithm, and §3 develops the composition phase. Section 4 combines these results to yield an algorithm for the consecutive-retrieval problem.

Undefined graph-theoretic terminology is consistent with Bondy and Murty [3]. With respect to the main algorithm and its subroutines described in the remaining sections, the data structures required are standard and so are not stressed in the derivation of the complexity bound. In particular, it is assumed that matrices are stored in a column-by-column fashion and that the nonzero positions for a given column are stored in a linked list. Graphs are stored by keeping the set of ends for each edge. In all stated complexity results, r is the number of rows of the given matrix, c is the number of columns, and n is the number of nonzeros.

2. A matrix decomposition. In this section, a matrix decomposition is described. This decomposition is first described independently of the consecutive-retrieval problem. Then the decomposition and corresponding composition results are related to the consecutive-retrieval problem.

A $\{0, 1\}$-matrix M is *standard* if $M = (I, N)$, where I denotes an identity matrix of appropriate size. Associated with a matrix M is a bipartite graph $B(M)$ defined as follows. The vertex set of $B(M)$ is $R(M) \cup C(M)$, where $R(M)$ denotes the index set of the rows of M and $C(M)$ denotes the index set of the columns of M, and $i \in R(M)$ and $j \in C(M)$ are adjacent in $B(M)$ if and only if the ij-entry of M is nonzero. The vertex set of a component of B induces a unique submatrix of M, called a *component* of M. The matrix M is *connected* if it has exactly one component.

Let M be a connected, standard $\{0, 1\}$-matrix, and let $B := B(M)$ be its associated bipartite graph. Let D be a proper subset of $C(M)$. If $B \setminus D$ is disconnected and no vertex of $R(M)$ is isolated in $B \setminus D$, then D is a *disconnecting set* (abbreviated *d-set*) of M. A d-set is *minimal* if it does not properly contain a d-set.

LEMMA 2.1. *Let $M = (I, N)$ be a connected $\{0, 1\}$-matrix. If M has at least two rows, then M has a minimal d-set. Moreover, every minimal d-set of M is contained in $C(N)$, and a minimal d-set can be computed in $O(n)$ time.*

Proof. Let $B := B(M)$, and let the vertex set of B be $R \cup C$, where $R := R(M)$ and $C := C(M)$. Let D be a minimal subset of C such that $B \setminus D$ is disconnected. (Since B is bipartite and M has at least two rows, such a set D exists.)

Suppose that D contains an element of $C(I)$, say u. Observe that u has degree one in B. Thus, $B \setminus (D - \{u\})$ is disconnected, a contradiction to the minimality of D. Thus, $D \subseteq C(N)$.

Consider any vertex $v \in R$ in $B \setminus D$. Since $D \subseteq C(N)$, v is adjacent, in $B \setminus D$, to a vertex in $C(I)$. Therefore, no vertex of R is isolated in $B \setminus D$. Thus, D is a d-set of M. Moreover, by the choice of D, it is a minimal d-set of M.

Now, let D be a minimal d-set of M. As above, if D contains an element u of $C(I)$, then $B \setminus (D - \{u\})$ is disconnected, which implies that $D - \{u\}$ is a d-set. Thus, every minimal d-set is contained in $C(N)$.

From the above, it follows that a minimal d-set of M can be found by computing a minimal subset D of C such that $B \setminus D$ is disconnected. This can be done as follows. Choose any $v \in R$. Let $u \in C$ index the identity column such that u and v are adjacent in B. Then u has degree one in B. Thus, v is a cut vertex of B. Let $B_1, \dots, B_t$ be the blocks (i.e., maximal 2-connected subgraphs) of B that contain v, and let $H_1, \dots, H_t$ be the maximal edge-disjoint connected subgraphs of B such that B_i is a subgraph of H_i for $1 \le i \le t$. Since M has at least two rows, it can be assumed without loss of generality that $|V(H_1)| \ge 3$. Let D' be the set of vertices of H_1 adjacent to v. Then D' is a d-set; it is a minimal d-set if $H_1 \setminus D'$ is connected. In the case that $H_1 \setminus D'$ is not connected, let K a component of $H_1 \setminus D'$ that has the fewest neighbors in D'; let D be this set of neighbors. Then D is the required set.

Constructing B requires $O(n)$ time by scanning each column of M, and finding the blocks of B requires $O(n)$ time using the algorithm of Aho, Hopcroft, and Ullman [1] designed for this purpose. Constructing the subgraphs $H_1, \dots, H_t$ and the set D' also takes $O(n)$ time. Finally, computing the components of $H_1 \setminus D'$ and constructing the set D require $O(n)$ time. □

Let D be a minimal d-set of a connected, standard matrix M. Let $R_1, \dots, R_t$ and $C_1, \dots, C_t$ be the partitions of $R(M)$ and $C(M) - D$ induced by the components of $B(M) \setminus D$. For $1 \le i \le t$, define M_i to be the submatrix of M having row set R_i and column set $C_i \cup D$. Then $\{M_1, \dots, M_t\}$ is a *simple decomposition* of M with respect to D. A matrix picture of a simple decomposition is given in Fig. 2.1. Observe that each M_i is a connected, standard matrix.

$$M = \left[\begin{array}{ccc|ccc|c} I_1 & & 0 & N_1 & & 0 & D_1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & & I_t & 0 & & N_t & D_t \end{array}\right] \qquad M_i = \left[\begin{array}{c|c|c} I_i & N_i & D_i \end{array}\right]$$

FIG. 2.1. *Matrix picture of a simple decomposition.*

Applying the notion of a simple decomposition recursively leads to the following definition. A *decomposition tree* associated with M is an arborescence $\mathcal{T}$ such that: (i) the vertex set of $\mathcal{T}$ is a collection of $\{0, 1\}$-matrices with the root of $\mathcal{T}$ equal to M, (ii) the set of children of a vertex is a simple decomposition of the vertex, and (iii) each vertex of outdegree zero is a matrix having exactly one row. Observe that associated with each vertex having outdegree greater than one is a minimal d-set, namely, the minimal d-set with respect to which the set of children of the vertex is a simple decomposition.

THEOREM 2.2. *Let M be a standard, connected $\{0, 1\}$-matrix. Then any decomposition tree associated with M has $O(r)$ vertices. Moreover, a decomposition tree can be computed in $O(rn)$ time.*

Proof. By Lemma 2.1, a minimal d-set can be computed in $O(n)$ time. Therefore, it suffices to show that any decomposition tree, say $\mathcal{T}$, associated with M has $O(r)$ vertices. This is done by showing by induction that $\mathcal{T}$ has at most $2r - 1$ vertices.

If $r = 1$, then $\mathcal{T}$ has exactly one vertex. Thus, assume $r \geq 2$. Let $\{M_1, \ldots, M_t\}$ be a simple decomposition of M, and let r_i be the number of rows of M_i for $1 \leq i \leq t$. By induction, any decomposition tree associated with M_i has at most $2r_i - 1$ vertices.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_t$ be the components of the arborescence obtained by deleting M from $\mathcal{T}$ such that M_i is a vertex of $\mathcal{T}_i$. Then $\mathcal{T}_i$ is a decomposition tree associated with M_i. Therefore, the tree $\mathcal{T}$ has at most $\sum_{i=1}^{t} (2r_i - 1) + 1$ vertices. Since $\sum_{i=1}^{t} r_i = r$ and $t \geq 2$, the result follows. □

The above decomposition is now related to cr-graphic matrices. Recall that the decomposition applies only to connected, standard matrices. Clearly, a matrix M is cr-graphic if and only if the standard matrix (I, M) is cr-graphic. Thus, nothing is lost by the restriction to standard matrices. Moreover, the following proposition reduces the analysis of the consecutive-retrieval problem to the connected case.

PROPOSITION 2.3. *Let M be a $\{0, 1\}$-matrix, and let $\{M_1, \ldots, M_t\}$ be the set of components of M. Then M is cr-graphic if and only if $M_1, \ldots, M_t$ are cr-graphic.*

Proof. Suppose that M is cr-graphic, and let T be a realization of M. Consider a component M_1 of M, and let T_1 be the subgraph of T induced by $R(M_1)$.

Let $c \in C(M_1)$. Then $c \in C(M)$. Let P be the vertex set of the path of T for which column c of M is the incidence vector. Since M_1 is a component of M, all of the vertices of P are vertices of T_1. Thus, P is the vertex set of a path in T_1, which implies that column c of M_1 is the incidence vector of P in T_1. Thus, if T_1 is a tree, then T_1 is a realization of M_1. Since T_1 is a subgraph of a tree, it is a tree if it is connected. If T_1 is not connected, then there exists a partition of its vertex set, say $\{V_1, V_2\}$, such that no vertex set of a path corresponding to a column of M_1 has nonempty intersection with both V_1 and V_2. It follows that M_1 is not connected, a contradiction.

Suppose that $M_1, \ldots, M_t$ are cr-graphic with respective realizations $T_1, \ldots, T_t$. Let v_i be a vertex of T_i for $1 \leq i \leq t$. Define T to be the tree obtained from $T_1, \ldots, T_t$ be adding $v_1v_2, v_2v_3, \ldots, v_{t-1}v_t$ as edges. Then the tree T is a realization of M. □

Let G be a connected graph, and let T be a spanning tree of G. Then the ordered pair (G, T) is a *graph-tree pair* (abbreviated *gt-pair*). For each edge e of G not in T, there exists a unique cycle in $T \cup \{e\}$, called a *fundamental cycle* of (G, T).

Let M be a connected, standard, cr-graphic matrix, and let T be a realization of M. For each column c, add an edge c joining the two ends of the corresponding path in T and call the resulting graph G. The gt-pair (G, T) is a *graph-tree realization* (abbreviated *gt-realization*) of M. Observe that in solving the consecutive-retrieval problem, it is sufficient to construct a gt-pair that is a gt-realization of the matrix.

The next sequence of results relates a simple decomposition of a matrix M to a gt-realization (G, T) of M.

LEMMA 2.4. *Let M be a cr-graphic matrix, and let (G, T) be a gt-realization of M. Then M is connected if and only if G is 2-edge connected.*

Proof. Suppose M is not connected. Then the associated bipartite graph is not connected. This implies that there exists a partition $\{V_1, V_2\}$ of the vertex set of G such that the vertex set of every fundamental cycle of (G, T) is contained in either V_1 or V_2. It follows that either G is disconnected or has a cut edge. Thus, G is not 2-edge connected.

By reversing the above argument, if G is not 2-edge connected, then M is not connected. □

Let H be a connected graph, and let K be the set consisting of the cut edges of H. Then, each component of $H \setminus K$ is a 2-*block* of H. Observe that each 2-block of H is 2-edge connected.

Let M be a connected, standard, cr-graphic matrix, and let (G, T) be a gt-realization of M. Lemmas 2.1 and 2.4 imply that a minimal d-set of M corresponds to a minimal subset D of $E(G) - T$ such that $G \setminus D$ has at least two 2-blocks.

LEMMA 2.5. *Let M be a connected, standard, cr-graphic matrix, and let (G, T) be a gt-realization of M. Let D be a minimal d-set of M, and let $H_1, \dots, H_t$ be the 2-blocks of $G \setminus D$. Let H be the graph obtained from G by contracting the edges of $\bigcup_{i=1}^{t} E(H_i)$, and let $P := H \setminus D$. Then P is a path with at least two vertices. Moreover, in H, the ends of every edge in D are coincident with the ends of P.*

Proof. Clearly, P is a tree with at least two vertices. Let J be a 2-block of $G \setminus D$ that corresponds to a vertex of degree one in P. Let D' be the subset of the edges of D that have an end in $V(J)$. Then D' is a d-set, and by minimality, $D' = D$. It follows that P has exactly two vertices of degree one and thus is a path. Moreover, every edge of D has its ends coincident, in H, with those of P. □

Let (G, T) be a gt-pair in which G is 2-edge connected, and let M be a matrix for which (G, T) is a gt-realization. Let D be subset of $E(G) - T$ such that D is a minimal d-set of M. Let $\{H_1, \dots, H_t\}$ be the 2-blocks of $G \setminus D$ such that H_1 and H_t correspond to the end vertices of the path specified by Lemma 2.5. Then H_1 and H_t each have exactly one vertex of attachment in $G \setminus D$. Consider H_1, and let x denote its vertex of attachment. By Lemma 2.5, each edge of D has exactly one of its ends in H_1. For edge $e \in D$, denote this end by $H_1(e)$. Let G_1 be the graph obtained from H_1 by adding the edges of D such that $e \in D$ has ends $H_1(e)$ and x in G_1. Define the graph G_t in an analogous fashion. Now, consider one of the remaining 2-blocks, say H_j for some $j \in \{2, \dots, t-1\}$. Then H_j has two vertices of attachment, say u and v, with the possibility that $u = v$. For each $j \in \{2, \dots, t-1\}$, define G_j to be the graph obtained from H_j by adding the edges of D such that every edge of D has ends u and v in G_j. Finally, for $1 \le i \le t$, define $T_i := T \cap E(G_i)$. Then $(G_1, T_1), \dots, (G_t, T_t)$ are the *D-components* of (G, T); (G_1, T_1) and (G_t, T_t) are the *end* D-components. Observe that if $\{M_1, \dots, M_t\}$ is a simple decomposition of M with respect to D, then for each $i \in \{1, \dots, t\}$, there exists a $j \in \{1, \dots, t\}$ such that $V(G_i) = R(M_j)$. Thus, by suitably ordering the $M_1, \dots, M_t$, it can be assumed that $V(G_i) = R(M_i)$ for $1 \le i \le t$.

LEMMA 2.6. *Let M be a connected, standard, cr-graphic matrix, and let $\{M_1, \dots, M_t\}$ be a simple decomposition of M with respect to D. Let (G, T) be a realization of M, and let $\{(G_1, T_1), \dots, (G_t, T_t)\}$ be the D-components of (G, T) with $V(G_i) = R(M_i)$ for $1 \le i \le t$. Then, for $1 \le i \le t$, (G_i, T_i) is a gt-realization of M_i.*

Proof. By Lemma 2.5, every edge of G not in T is either an edge of D or has both of its ends in G_i for some $i \in \{1, \dots, t\}$. Also by Lemma 2.5 and the definition of the D-components, for a given edge e of G_i not in T_i, the vertex set of the fundamental cycle of

(G_i, T_i) that contains e is the restriction of the vertex set of the fundamental cycle of (G, T) that contains e to the vertex set of G_i. From the definition of simple decomposition, it follows that each column of M_i is the restriction of a column of M to the row set of M_i. The result follows. □

A set D of edges in a graph G is a *single star* if there exists at least one vertex of G to which all of the edges of D are incident; each such vertex is an *end* of D. Note that some of the edges in D might be loops. A set D of edges in a graph G is a *double star* if there exist at most two vertices of G to which the edges of D are incident. Note that these two vertices can be identical. Observe that every double star is also a single star. Also, observe that in the definition of the D-components above, the set D is a single star in G_1 and G_t and is a double star in $G_2, \ldots, G_{t-1}$.

The composition of gt-pairs is now examined. Let $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ be a set of gt-pairs, and let D be a subset of $\bigcap_{i=1}^{t} E(G_i)$. Then $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is *mergeable* with respect to D if: (i) $V(G_i) \cap V(G_j) = \emptyset$ for $1 \leq i < j \leq t$, (ii) $D \cap T_i = \emptyset$ for $1 \leq i \leq t$, (iii) D is a single star in $G_1, \ldots, G_t$, and (iv) D is a double star in at least $t-2$ of $G_1, \ldots, G_t$.

Consider a mergeable set of gt-pairs with respect to a set D. An ordering of the members of this mergeable set is *feasible* if D is a double star in the second through the penultimate gt-pairs (actually, the graphs of the gt-pairs) of the ordering. Every mergeable set has a feasible ordering.

Let $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ be a mergeable set with respect to a set D, and suppose $(G_1, T_1), \ldots, (G_t, T_t)$ is a feasible ordering. A sequence $X := x_1, \ldots, x_{2t-2}$ is a *merging sequence* for $(G_1, T_1), \ldots, (G_t, T_t)$ and D if D is a single star of (G_1, T_1) with end x_1, D is a single star of (G_t, T_t) with end x_{2t-2}, and, for $2 \leq i \leq t-1$, D is a double star of (G_i, T_i) with ends x_{2i-2} and x_{2i-1}. Note that for a given feasible ordering, there can exist several merging sequences. However, given a mergeable set and a merging sequence, there exists exactly one corresponding feasible ordering. Thus, it is well defined to associate a merging sequence directly with a mergeable set.

Let $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ be a mergeable set with respect to a set D, and let $X = x_1, \ldots, x_{2t-2}$ be an associated merging sequence such that x_1 is a vertex of G_1, x_{2t-2} is a vertex of G_t, and x_{2i-2} and x_{2i-1} are vertices of G_i for $2 \leq i \leq t-1$. A gt-pair (G, T) is constructed as follows. For $f \in D$, if f is not a loop of G_1, then let $u(f)$ denote the end of f different from x_1; otherwise, let $u(f) = x_1$. For $f \in D$, if f is not a loop of G_t, then let $v(f)$ denote the end of f different from x_{2t-2}; otherwise, let $v(f) = x_{2t-2}$. Define G to be the graph obtained from $G_1 \setminus D, \ldots, G_t \setminus D$ by adding the edges of D such that $f \in D$ has ends $u(f)$ and $v(f)$ and by adding new edges $e_1, \ldots, e_{t-1}$ such that e_i has ends x_{2i-1} and x_{2i}. Define $T := \{\bigcup_{i=1}^{t} T_i\} \cup \{\bigcup_{i=1}^{t-1} \{e_i\}\}$. Then (G, T) is the *mergence* of $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ with respect to D and X. Observe that $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is precisely the set of D-components of (G, T).

LEMMA 2.7. *Let M be a standard, connected, cr-graphic matrix, and let $\{M_1, \ldots, M_t\}$ be a simple decomposition of M with respect to a minimal d-set D. For $1 \leq i \leq t$, let (G_i, T_i) be a gt-realization of M_i. Suppose $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is mergeable with respect to D. Then, the mergence of $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ with respect to D and any merging sequence is a gt-realization for M.*

Proof. Let (G, T) denote the mergence of $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ with respect to D and some merging sequence.

Consider a column d of M. If $d \notin D$, then d is exactly in one M_i, so by Lemma 2.6, column d of M_i is the incidence vector of the vertex set of a fundamental cycle, say C_d, in (G_i, T_i). Observe that the mergence maintains C_d, and thus, C_d is a fundamental cycle in (G, T). If $d \in D$, then d is partitioned into t subcolumns such that subcolumn d_i is in exactly one M_i. Moreover, each d_i is the incidence vector of the vertex set of some fundamental cycle C_{d_i} of (G_i, T_i). Again, from the definition of the mergence, $\bigcup_{i=1}^{t} V(C_{d_i})$ is the vertex set of a fundamental cycle of (G, T) whose corresponding incidence vector is d. Thus, every column of M is the incidence vector of the vertex set of a fundamental cycle of (G, T).

Observe that the number of fundamental cycles of (G, T) is equal to the number of columns of M. It follows that the vertex set of every fundamental cycle of (G, T) has its incidence vector as a column of M. □

Lemmas 2.6 and 2.7 can be combined to yield the following result, which motivates the algorithm for solving the consecutive-retrieval problem.

THEOREM 2.8. *Let M be a connected, standard matrix, and let $\{M_1, \ldots, M_t\}$ be a simple decomposition of M with respect to a minimal d-set D. Then M is cr-graphic if and only if $M_1, \ldots, M_t$ are cr-graphic and have gt-realizations such that the set of these gt-realizations is mergeable with respect to D.*

3. Composition. The purpose of this section is to develop the theory used to carry out the composition phase of the algorithm. The idea is as follows. Let M be a connected, standard $\{0, 1\}$-matrix, and suppose that it is desired to determine whether M is cr-graphic. Let $\mathcal{T}$ be a decomposition tree of M. Each vertex of $\mathcal{T}$ of outdegree zero is a matrix having exactly one row. Such matrices are evidently cr-graphic; in fact, they have unique gt-realizations. At a general stage of the composition phase, a vertex N of $\mathcal{T}$ is considered. Suppose that all of the children of N in $\mathcal{T}$ have been found to be cr-graphic, and let $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ be gt-realizations for the children of N. If $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is mergeable, then, by Lemma 2.7, a gt-realization of N can be obtained by first choosing a merging sequence and then constructing the resulting mergence. In this case, this process can be repeated at another vertex of $\mathcal{T}$. However, it might be the case that $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is not mergeable. Unfortunately, if this happens, then it does not necessarily follow that N is not cr-graphic. It might be the case that $\{(G_1, T_1), \ldots, (G_t, T_t)\}$ is an unsuitable set of gt-realizations for the children of N. That is, there might exist another set of gt-realizations of the children of N that is mergeable. In particular, if M is cr-graphic, then it follows from Lemma 2.6 that a mergeable set of gt-realizations exists. Thus, it is sufficient to find a mergeable set of gt-realizations of the children of N or to show that no such set exists.

Observe that the choice of the set of gt-realizations of the children of N (and the corresponding merging sequence) not only determines the gt-realization of N but also affects those of the ancestors of N in $\mathcal{T}$. Therefore, in order to make a judicious choice for the gt-realizations for the children of N, one must take into account information about the minimal d-sets associated with N and all of its ancestors in $\mathcal{T}$. The following set of definitions are aimed at making these ideas precise.

Let M be a connected, standard matrix, and let $\mathcal{T}$ be a decomposition tree of M. Let N be a vertex of $\mathcal{T}$ different from M, and let P be the (M, N)-dipath of $\mathcal{T}$. Let $M_1, \ldots, M_m$ be the vertex set of P in the order specified by P. Then M_{i+1} is a member of a simple decomposition of M_i with respect to some minimal d-set of M_i, say D_i. Define the ordered set $\Sigma := (D_1, \ldots, D_{m-1})$ to be the *decomposer* of N with respect to $\mathcal{T}$. The *critical decomposer* of N is the subset $\{D_i \in \Sigma | D_i \subseteq C(N)$ and $D_i - D_j \neq \emptyset$ for every $m-1 \geq j > i\}$. The *decomposer* and *critical decomposer* of M are defined to be the empty set.

Observe that if N' is the parent of N in $\mathcal{T}$, then $(D_1, \ldots, D_{m-2})$ is the decomposer of N'. Moreover, if Ω is the critical decomposer of N, then the critical decomposer of N' is obtained from Ω by first deleting D_{m-1} and then adding the members of $\{D_i \in \Sigma | D_i \subseteq D_{m-1}$ and $D_i - D_j \neq \emptyset$ for every $m-2 \geq j > i\}$ and the members of $\{D_i \in \Sigma | D_i - C(N) \neq \emptyset, D_i \subseteq C(N')$, and $D_i - D_j \neq \emptyset$ for every $m-2 \geq j > i\}$.

Let (G, T) be a gt-pair. Let $\Omega = \{D_1, \dots, D_m\}$ be a set of subsets of $E(G) - T$ such that, for $1 \leq i \leq m$, D_i is a single star of G. Define $V_\Omega := \{v \in V(G) | v$ is an end of some $D_i \in \Omega\}$. A *labeling* of (G, T) with respect to Ω is a function L that maps elements of V_Ω into subsets of Ω such that each $D_i \in L(v)$ for some v and if $D_i \in L(v)$, then v is an end of D_i. If $D_i \in L(v)$ for some $v \in V_\Omega$, then v is D_i*-labeled* by L. A labeling L is *consistent* with respect to Ω if it satisfies properties (C1) and (C2) below.

(C1) For every $1 \leq i < j \leq m$, if $e \in D_i \cap D_j$ and e is not a loop, then no end of e is both D_i-labeled and D_j-labeled by L.

(C2) For every $1 \leq i \leq m$, if D_i is a double star and $D_i \cap D_j = \emptyset$ for all $j \neq i$, then both ends of D_i are D_i-labeled by L.

If (G, T) has a consistent labeling with respect to Ω, then (G, T) is *consistent* with respect to Ω.

In the context of the composition phase of the algorithm described at the beginning of this section, consistent labelings are used as follows. As before, let N be a vertex of the decomposition tree $\mathcal{T}$ associated with M, and let $\{(G_1, T_1), \dots, (G_t, T_t)\}$ be gt-realizations of the children of N. Assume that each (G_i, T_i) has a consistent labeling, say L_i, with respect to the critical decomposer of the corresponding vertex in $\mathcal{T}$. Then $L_1, \dots, L_t$ are used to construct an appropriate merging sequence for $\{(G_1, T_1), \dots, (G_t, T_t)\}$. In particular, it will be seen that if D is the minimal d-set associated with N and $D \in L_i(x)$ for some $x \in V(G_i)$, then x is a good vertex to put in the merging sequence.

The next result says that if M is cr-graphic, then a consistent labeling does, in fact, exist.

LEMMA 3.1. *Let M be a connected, standard, cr-graphic matrix, and let $\mathcal{T}$ be a decomposition tree of M. Let N and N' be vertices of $\mathcal{T}$ such that N' is a child of N, and let D be the minimal d-set associated with N. Let Ω and Ω' be the respective critical decomposers of N and N'. Let (H, S) and (H', S') be respective gt-realizations of N and N' such that (H', S') is a D-component of (H, S). If (H, S) is consistent with respect to Ω, then (H', S') is consistent with respect to Ω'.*

Proof. Since (H, S) is consistent with respect to Ω, every member of Ω is a single star of H. Observe that $\Omega' \subseteq \Omega \cup \{D\}$. From the definition of the D-components, it follows that every member of Ω' is a single star of H'. Now, let L be a consistent labeling of (H, S) with respect to Ω. It suffices to construct a consistent labeling, say L', of (H', S') with respect to Ω'. This is done by specifying $L'(v)$ for each vertex v of H' that is an end of some member of Ω' in H'. Let v be such a vertex, and choose $D' \in \Omega'$ such that v is an end of D' in H'. If $D' \neq D$ and $D' \in L(v)$, then define D' to be a member of $L'(v)$. If $D' = D$ and v is not an end of D in H, then define D to be a member of $L'(v)$. Finally, if $D' = D$, v is an end of D in H, and either D is a set of loops of H' incident to v or D is a double star of H' and $D \cap D'' = \emptyset$ for all $D'' \in \Omega' - \{D\}$, then define D to be a member of $L'(v)$.

It is claimed that L' defined above is a consistent labeling of (H', S') with respect to Ω'. The first step is to show that L' is a labeling. Let $D' \in \Omega' - \{D\}$. It follows from the definition of the D-components that every end of D' in H is also an end of D' in H'. Since L is a labeling, it follows that $D' \in L(v)$ for some end v of D' in H'. From the definition of L', $D' \in L'(v)$. Also from the definition of L' and the D-components, it follows that $D \in L'(v)$ for some end v of D in H'. Thus, L' is a labeling.

The next step is to show that (C1) and (C2) are satisfied by L' and Ω'. Consider (C1), and let D' and D'' be members of Ω' that have a common edge, say $e = uv$, that is not a loop of H'. Moreover, assume $L'(v)$ contains both D' and D''. Since e is not a loop of H', it is not a loop of H. Assume, without loss of generality, $D'' \neq D$. Since $D'' \in L'(v)$, it follows from the definition of L' that $D'' \in L(v)$. Thus, v is an end of D'' in H, which implies that v is an end of e in H. If $D' \neq D$, then similarly $D' \in L(v)$, which is a contradiction to (C1).

Thus, $D' = D$. Now, from the definition of L', it follows that v is not an end of D in H. The definition of the D-components together with the facts that v is an end of e in H, v is an end of D in H', but v is not an end of D in H imply that e is a loop of H', a contradiction.

Finally, consider (C2). Let D' be a member of Ω' such that D' is a double star in H' and $D' \cap D'' = \emptyset$ for all $D'' \in \Omega' - \{D'\}$. If $D' = D$, then the definition of L' implies that both ends of D' in H' are D'-labeled, as required. If $D' \neq D$, then D' is a double star of H and $D' \cap D'' = \emptyset$ for $D'' \in \Omega - \{D'\}$. Since L is consistent, both ends of D' in H are D'-labeled by L. It follows that both ends of D' in H' are D'-labeled by L', as required. □

The next set of results is aimed at proving the converse of Lemma 3.1. For this purpose, the following notations are fixed for the remainder of the section. Let M be a connected, standard $\{0, 1\}$-matrix, and let $\mathcal{T}$ be a decomposition tree of M. Let N be a vertex of $\mathcal{T}$, and let $N_1, \dots, N_t$ be the children of N in $\mathcal{T}$. Let D be the minimal d-set associated with N, and let $\Omega, \Omega_1, \dots, \Omega_t$ be the critical decomposers of $N, N_1, \dots, N_t$, respectively. Throughout the remainder of the section, $N_1, \dots, N_t$ are assumed to be cr-graphic and have gt-realizations that are consistent with respect to their respective critical decomposers. In particular, for $1 \leq i \leq t$, let (H_i, S_i) denote a gt-realization of N_i, and let L_i be a consistent labeling of (H_i, S_i) with respect to Ω_i. On occasion, N is assumed to be cr-graphic. In this case, (G, T) denotes a gt-realization of N and $(G_1, T_1), \dots, (G_t, T_t)$ are the D-components of (G, T). By Lemma 2.6, $(G_1, T_1), \dots, (G_t, T_t)$ are also gt-realizations of $N_1, \dots, N_t$. By the definition of the D-components, D is a double star in at least $t - 2$ of $G_1, \dots, G_t$.

Under the assumption that M is cr-graphic, Lemmas 3.2 to 3.4 establish the existence of a special type of merging sequence for $\{(H_1, S_1), \dots, (H_t, S_t)\}$ with respect to $L_1, \dots, L_t$ and D. Lemmas 3.5–3.8 then show that the resulting mergence is consistent with respect to Ω. This is done algorithmically; Procedures LABEL1, LABEL2, and LABEL3 construct a consistent labeling from $L_1, \dots, L_t$.

LEMMA 3.2. *If N is cr-graphic, then $\{(H_1, S_1), \dots, (H_t, S_t)\}$ is mergeable.*

Proof. By assumption, each (H_i, S_i) is consistent with respect to Ω_i. From the definition of consistency, D is a single star in each of $H_1, \dots, H_t$. Thus, it suffices to show that D is a double star in at least $t - 2$ of $H_1, \dots, H_t$.

As observed, D is a double star in at least $t - 2$ of $G_1, \dots, G_t$. Since (H_i, S_i) and (G_i, T_i) are both gt-realizations of the matrix N_i, it follows that if D is a double star in G_i, then it is a double star of H_i. □

For $1 \leq i \leq t$, define (H_i, S_i) to be an *end piece* of $\{(H_1, S_1), \dots, (H_t, S_t)\}$ with respect to D if either D is a single star and not a double star of H_i or D is a double star of H_i and $D' \cap D \neq \emptyset$ for some $D' \in \Omega_i - \{D\}$.

LEMMA 3.3. *If M is cr-graphic, then at most two of* $\{(H_1, S_1), \dots, (H_t, S_t)\}$ *are end pieces with respect to D.*

Proof. Since M is cr-graphic, repeated application of Lemma 2.6 implies that N is cr-graphic with gt-realization (G, T). It is shown that if (H_i, S_i) is an end piece of $\{H_1, \dots, H_t\}$, then (G_i, T_i) is an end D-component of (G, T). Since (G, T) has exactly two end D-components, the result follows.

Suppose that (H_i, S_i) is an end piece for some $i \in \{1, \dots, t\}$. If D is a single star and not a double star of H_i, then D is a single star and not a double star of G_i, which implies that (G_i, T_i) is an end D-component of (G, T). Now suppose that D is a double star of H_i and that $D' \cap D \neq \emptyset$ for some $D' \in \Omega_i - \{D\}$. Since Ω_i is a critical decomposer, $D' - D \neq \emptyset$. Since $D' \in \Omega_i$, D' is a single star of G_i. Since $D' - D \neq \emptyset$, there exists an edge e of $D' - D$ that, in G, has both ends in $V(G_i)$. Since $D' \cap D \neq \emptyset$, there exists an edge f of G in $D' \cap D$. By repeated application of Lemma 3.1, (G, T) is consistent with respect to Ω. Since $D' \in \Omega$, it is a single star of G. Thus, e and f are adjacent in G. It follows that (G_i, T_i) is an end D-component of (G, T). □

A feasible ordering of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ is *good* if none of the second through penultimate gt-pairs in the ordering is an end piece of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$. If M is cr-graphic, then Lemma 3.3 implies that $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ has a good ordering. A merging sequence $X = x_1, \ldots, x_{2t-2}$ for a good ordering of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ is *good* with respect to $L_1, \ldots, L_t$ and D if $D \in L_i(x_j)$ whenever $x_j \in V(H_i)$. As before, it is well defined to associate a good merging sequence directly with a mergeable set; the associated unique ordering of the mergeable set is necessarily good. The next result shows that there exists a good merging sequence.

LEMMA 3.4. *If M is cr-graphic, then there exists a good merging sequence for $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ with respect to $L_1, \ldots, L_t$ and D.*

Proof. By Lemma 3.3, it can be assumed that $(H_1, S_1), \ldots, (H_t, S_t)$ is a good ordering of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$. Since L_1 is consistent, there exists a vertex $x_1 \in V(H_1)$ such that $D \in L_1(x_1)$. Similarly, there exists a vertex $x_{2t-2} \in V(H_t)$ such that $D \in L_t(x_{2t-2})$. Now, consider $i \in \{2, \ldots, t-1\}$. Since $(H_1, S_1), \ldots, (H_t, S_t)$ is a good ordering, D is a double star of H_i. Moreover, since (H_i, S_i) is not an end piece, $D \cap D' = \emptyset$ for all $D' \in \Omega_i - \{D\}$. If the ends of D in H_i are x_{2i-2} and x_{2i-1}, then property (C2) of the consistency of L_i implies that $D \in L_i(x_{2i-1})$ and $D \in L_i(x_{2i-2})$. Thus, the sequence $x_1, \ldots, x_{2t-2}$ is a good merging sequence. □

The remainder of the section assumes that $\{H_1, \ldots, H_t\}$ is mergeable. If M is cr-graphic, then Lemmas 2.6 and 3.2 imply that this is indeed the case. The following notations are fixed: X denotes a merging sequence of $\{(H_1, S_1), \ldots, (H_, S_t)\}$ with respect to $L_1, \ldots, L_t$ and D, and (H, S) is the mergence of $\{(H_1, S_1), \ldots, (H_, S_t)\}$ with respect to D and X. In the case that M is indeed cr-graphic, it is further assumed that X is a good merging sequence, the existence of which is provided by Lemma 3.4. Finally, it is also assumed that the ordering of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ corresponding to X is given by $(H_1, S_1), \ldots, (H_t, S_t)$. The next lemma establishes a necessary condition for the existence of a consistent labeling of (H, S) with respect to Ω.

LEMMA 3.5. *If M is cr-graphic, then every member of Ω is a single star in H.*

Proof. Let $D' \in \Omega$. Since M is cr-graphic, Lemma 2.6 implies that N is also cr-graphic. Moreover, Lemma 3.1 implies that (G, T) is consistent with respect to Ω. Thus, D' is a single star of G. It follows that $D' \subseteq E(G_i)$ for some $i \in \{1, \ldots, t\}$. Thus, either $D' \in \Omega_i$ for some $i \in \{1, \ldots, t\}$ or $D' \subseteq D$.

First, suppose that $D' \in \Omega_i$ for some $i \in \{1, \ldots, t\}$. Since L_i is a consistent labeling of (H_i, S_i) with respect to Ω_i, there exists a vertex $v \in V(H_i)$ such that $D' \in L_i(v)$. Moreover, v is an end of D' in H_i. If $D \cap D' = \emptyset$, then it follows directly from the definition of the mergence that D' is a single star in H with v as an end. Thus, assume $D \cap D' \neq \emptyset$. Then (H_i, S_i) is an end piece. Moreover, by (C1), either $D \notin L_i(v)$ or every edge of $D \cap D'$ is a loop of H_i. In either case, from the definition of the mergence, it follows that D' is a single star in H.

Next, suppose that $D' \subseteq D$. Let v be an end of D' in G. Then v is a vertex of one of the end D-components of (G, T), say (G_j, T_j) for some $j \in \{1, \ldots, t\}$. It follows that D' is a double star of G_j. By definition, D, and therefore D', is a double star in all of the nonend D-components of (G, T). Thus, D' is a double star in at least $t-1$ of $G_1, \ldots, G_t$, which implies that D' is a double star in at least $t-1$ of $H_1, \ldots, H_t$. Now, from the definition of the mergence, D' is a single star in H. □

The construction of a consistent labeling of (H, S) with respect to Ω is done in three stages. Define $\Sigma_1 := (\Omega_1 \cup \cdots \cup \Omega_t) - \{D\}$. Define Σ_2 to be those members of Ω that are contained in D and are not a double star in H. (By Lemma 3.5, each member of Σ_2 is a single star in H.) Finally, define Σ_3 to be those members of Ω that are contained in D and are a double star in H. Then $\Omega = \Sigma_1 \cup \Sigma_2 \cup \Sigma_3$. Moreover, Σ_1, Σ_2, and Σ_3 are pairwise disjoint. Define $\{\Sigma_1, \Sigma_2, \Sigma_3\}$ to be the *principal partition* of Ω. A consistent labeling of (H, S) with respect to Ω is constructed by first constructing a consistent labeling of (H, S) with respect to Σ_1, then to $\Sigma_1 \cup \Sigma_2$, and finally to $\Sigma_1 \cup \Sigma_2 \cup \Sigma_3$. The following procedure provides the first construction.

```
Procedure LABEL1.
Input: $(H, S)$, $\Sigma_1$, and $L_1, \dots, L_t$.
Output: A labeling $L$ of $(H, S)$ with respect to $\Sigma_1$.
Step 1 : For each $v \in V(H)$ that is an end of some member of $\Sigma_1$, set
    $L(v) \leftarrow \emptyset$.
Step 2 : For each $D' \in \Sigma_1$ and each $v \in V(H)$ that is an end of $D'$, if
    $D' \in L_i(v)$ for some $i \in \{1, \dots, t\}$, then set $L(v) \leftarrow L(v) \cup \{D'\}$.
```

LEMMA 3.6. *The output L of Procedure* LABEL1 *is a labeling of (H, S) with respect to Σ_1. Moreover, if M is cr-graphic, then L is consistent.*

Proof. Since $L_1, \dots, L_t$ are labelings, it follows that L is a labeling. Thus, assuming that M is cr-graphic, it suffices to show that properties (C1) and (C2) are satisfied.

Consider (C1), and let D' and D'' be members of Σ_1 such that D' and D'' have a nonloop edge $e = uv$ of H in common. Now, suppose that D' and D'' are both members of $L(v)$. Thus, v is an end of D' and D'' in H. It follows that there exists an $i \in \{1, \dots, t\}$ such that D' and D'' are members of Ω_i. From the definition of L, it follows that D' and D'' are both members of $L_i(v)$. Since L_i is a consistent labeling of (H_i, S_i) with respect to Ω_i, it follows that e is a loop of H_i. Thus, $e \in D$ and $i \in \{1, t\}$. Without loss of generality, assume $i = 1$.

Since D' and D'' are in Ω_1, $D' - D$ and $D'' - D$ are nonempty. Since e is a loop at v in H_1, it is a loop at v in G_1. Therefore, v is an end of both D' and D'' in G_1. It now follows from the definition of D-components of (G, T) that v is the unique end of both D' and D'' in G. Thus, in any labeling of (G, T) with respect to Ω, D' and D'' violate property (C2). Thus, (G, T) has no consistent labeling with respect to Ω, contradicting Lemma 3.1.

Now, consider (C2). Let $D' \in \Sigma_1$ be such that D' is a double star of H and $D' \cap D'' = \emptyset$ for all $D'' \in \Sigma_1 - \{D'\}$. Observe that $D' \cap D = \emptyset$. Thus, D' is a double star in H_i for some $i \in \{1, \dots, t\}$. Moreover, D' has the same set of ends in H_i as it does in H. Since $\Omega_i - \{D\} \subseteq \Sigma_1$, it follows that $D' \cap D'' = \emptyset$ for all $D'' \in \Omega_i - \{D'\}$. Therefore, by (C2), both ends of D' in H_i are D'-labeled by L_i. By Procedure LABEL1, both ends of D' in H are D'-labeled by L, as required. □

The following procedure extends the labeling L to a consistent labeling with respect to $\Sigma_1 \cup \Sigma_2$.

```
Procedure LABEL2.
Input: $(H, S)$, $\Sigma_2$, and the labeling $L$ that is the output of LABEL1.
Output: A labeling $L$ of $(H, S)$ with respect to $\Sigma_1 \cup \Sigma_2$.
Step 1 : For each $v \in V(H)$ that is an end of some member of $\Sigma_2$ but not
    an end of any member of $\Sigma_1$, set $L(v) \leftarrow \emptyset$.
Step 2 : For each $D' \in \Sigma_2$, let vertex $v$ be its unique end in $H$ and set
    $L(v) \leftarrow L(v) \cup \{D'\}$.
```

LEMMA 3.7. *The output L of Procedure* LABEL2 *is a labeling of (H, S) with respect to $\Sigma_1 \cup \Sigma_2$. Moreover, if M is cr-graphic, then L is consistent.*

Proof. Since the input to Procedure LABEL2 is a labeling, it is evident that the output is a labeling. Thus, assuming that M is cr-graphic, it suffices to show that (C1) and (C2) are satisfied. Since (C2) is satisfied by the input to the procedure, it is also satisfied by the output because no member of Σ_2 is a double star. Thus, consider (C1).

Suppose that there exist D' and D'' in $\Sigma_1 \cup \Sigma_2$ having a nonloop edge e of H in common, and suppose that D' and D'' are both members of $L(v)$, where v is an end of e in H. By Lemma 3.6, at most one of D' and D'' is in Σ_1. Without loss of generality, assume $D'' \notin \Sigma_1$.

First, consider the case when $D' \in \Sigma_1$. Since $D'' \in \Sigma_2$, $D'' \subseteq D$. Since v is an end of e in H and $e \in D$, v is a vertex of either H_1 or H_t. Without loss of generality, assume the former. Since D'' is not a double star in H, there exists an edge $f \in D''$ in H that is not parallel to e. Since $D' \in L(v)$, v is an end of D' in H. Thus, $D' - D$ is contained in the edge set of H_1. Observe that e and f are parallel in H_1. A contradiction is obtained by showing that they are not parallel in G_1. By Lemma 3.1, (G, T) is consistent with respect to Ω. Thus, D' and D'' are single stars of G. Let e have ends x and y in G. Since $D' - D \subseteq E(H_1)$, $D' - D \subseteq E(G_1)$. Since $D' \in \Sigma_1$, $D' - D \neq \emptyset$. It follows that (G_1, T_1) is an end D-component of (G, T), and x (say) is a vertex of G_1. Moreover, y is a vertex of the end D-component of (G, T) different from (G_1, T_1). Therefore, x is the unique end of D' in G and so is D'-labeled by any consistent labeling of (G, T) with respect to Ω. By consistency, x is not D''-labeled by any such labeling. Therefore, y is D''-labeled by any consistent labeling of (G, T) with respect to Ω. Thus, y is an end of D'' in G. Since e and f are not parallel in H, they are not parallel in G. Therefore, x is not an end of f in G. It follows that e and f are not parallel in G_1, as required.

Second, consider the case that D' and D'' are both members of Σ_2. Thus, both are contained in D and neither is a double star of H. Let f be an edge of D' that is not parallel to e in H, and let g be an edge of D'' that is not parallel to e in H. By the consistency of (G, T) with respect to Ω, the respective ends of D' and D'' in G are in different end D-components of (G, T). It follows that f and g are not parallel in either end D-component of (G, T). Thus, f and g are parallel in precisely $t - 2$ of the D-components of (G, T). This is a contradiction since f and g are parallel in $t - 1$ of $\{(H_1, S_1), \dots, (H_t, S_t)\}$; in particular, f and g are parallel in each of $\{(H_2, S_2), \dots, (H_{t-1}, S_{t-1})\}$ and the member of $\{(H_1, S_1), (H_t, S_t)\}$ that contains the vertex v. □

The final step is to extend the output L of Procedure LABEL2 to a consistent labeling of (H, S) with respect to Ω. The idea is as follows. Recall that each member of Σ_3 is a double star of H contained in D. Consider a graph $I := I(H, \Sigma_3)$, the vertex set of which is Σ_3 and the edge set of which is defined by declaring two members of Σ_3 to be adjacent if they have an edge of H in common. Consider a component J of I. Then there exist distinct vertices of H, say u and v, such that every member of Σ_3 that is a vertex of J is a double star with ends u and v. Now, suppose that the consistent labeling L of (H, S) with respect to $\Sigma_1 \cup \Sigma_2$ has been extended to $\Sigma_1 \cup \Sigma_2 \cup \{D'\}$ for some $D' \in V(J)$, and suppose that under this extension $D' \in L(u)$. Consider a member $D'' \in V(J)$ that is adjacent to D' in J. Then, if L is extended to include D'', it must be that $D'' \in L(v)$, for otherwise L would violate (C1). More generally, if $D'' \in V(J) - \{D'\}$ and some path in J from D' to D'' has an odd number of edges, then $D'' \in L(v)$; otherwise $D'' \in L(u)$. (It follows that if a consistent labeling exists, then the graph I is bipartite.) The following procedure makes these ideas precise. For any two vertices D' and D'' in the same component of I, let $d(D', D'')$ denote the number of edges in a shortest path from D' to D'' in I.

```
Procedure LABEL3.
Input: $(H, S)$, $\Sigma_3$, and the labeling $L$ that is the output of LABEL2.
Output: A labeling $L$ for $(H, S)$ with respect to $\Omega$.
Step 1: For each $v \in V(H)$ that is an end of some member of $\Sigma_3$ but not an
    end of any member of $\Sigma_1 \cup \Sigma_2$, set $L(v) \leftarrow \emptyset$.
Step 2 : Compute $I := I(H, \Sigma_3)$, and let $\mathcal{I}$ be the set of components of $I$.
Step 3 : If $\mathcal{I} = \emptyset$, then output $L$ and stop; otherwise, choose $J \in \mathcal{I}$.
```

Step 4 : Determine whether there exists a vertex D' of J that has an edge in common with a member D^* of $\Sigma_1 \cup \Sigma_2$; if so, let u' be an end of D' such that $D^* \notin L(u')$. Otherwise, let D' be an arbitrary vertex of J, and let u' be an end of D'. In either case, let v' be the end of D' different from u'.
Step 5 : For each $D'' \in V(J)$ such that $d(D', D'')$ is even, set $L(u') \leftarrow L(u') \cup \{D''\}$. For each $D'' \in V(J)$ such that $d(D', D'')$ is odd, set $L(v') \leftarrow L(v') \cup \{D''\}$. If D' is the unique vertex of J and D' does not have an edge in common with any member of $\Sigma_1 \cup \Sigma_2$, then set $L(v') \leftarrow L(v') \cup \{D'\}$. Set $\mathcal{I} \leftarrow \mathcal{I} - \{J\}$, and go to Step 3.

LEMMA 3.8. *The output L of Procedure* LABEL3 *is a labeling of* (H, S) *with respect to* Ω. *Moreover, if M is cr-graphic, then L is consistent.*

Proof. The first step is to verify that the output L is a labeling of (H, S) with respect to Ω. In particular, it must be verified that each member D' of Ω is in $L(v)$ for some vertex $v \in V(H)$ that is an end of D'. By Lemma 3.7, this is true for $D' \in \Sigma_1 \cup \Sigma_2$. Now, consider Σ_3, and in particular, consider a component J of the graph I computed in Step 2. The key point is to show that the vertex u' defined in Step 4 exists. Suppose that there exists $D' \in V(J)$ and $D^* \in \Sigma_1 \cup \Sigma_2$ such that D' and D^* have an edge in common. Since $D^* \in \Sigma_1 \cup \Sigma_2$, either D^* is a single star and not a double star or it is a double star not contained in D. Since D' is a double star contained in D, it has an end that is not an end of D^*. Therefore, the vertex u' exists. Given that u' exists, then Step 5 evidently assigns each vertex D'' of J to some appropriate $L(v)$. Therefore, L is a labeling of (H, S) with respect to Ω.

The next step is to verify the consistency of L in the case that M is cr-graphic. By Lemma 3.7 and Step 5, property (C2) is evidently satisfied. Consider property (C1). First, observe that it is sufficient to show that property (C1) is satisfied by L for pairs of members from $\Sigma_1 \cup \Sigma_2 \cup V(J)$ for each component J of I.

Let J be a component of I, and let u and v be the ends of some member (and therefore, every member) of $V(J)$ in H. Let x and y be the analogous vertices in G. Since every member of $V(J)$ is contained in D, $u \in V(H_1)$ (say) and $v \in V(H_t)$. Similarly, x and y are contained in the respective vertex sets of the end D-components of (G, T). Without loss of generality, assume that if (G_1, T_1) is an end D-component of (G, T), then $x \in V(G_1)$, and if (G_t, T_t) is an end D-component, then $y \in V(G_t)$. By Lemma 3.1, (G, T) has a consistent labeling with respect to Ω, say L'.

First, suppose that no member of $V(J)$ has an edge of H in common with a member of $\Sigma_1 \cup \Sigma_2$. Then, by Lemma 3.7, L satisfies property (C1) with respect to $\Sigma_1 \cup \Sigma_2 \cup V(J)$ if and only if it satisfies property (C1) with respect to $V(J)$. Let D' be the member of $V(J)$ chosen in Step 4. Observe that the graph I is identical to the analogous graph for G and Σ_3; that is, $I = I(G, \Sigma_3)$. Observe that if $D' \in L'(x)$ (say), then, for any $D'' \in V(J)$, $D'' \in L'(y)$ if and only if $d(D', D'')$ is odd. It follows that either $L(u) \cap V(J) = L'(x) \cap V(J)$ and $L(v) \cap V(J) = L'(y) \cap V(J)$ or $L(u) \cap V(J) = L'(y) \cap V(J)$ and $L(v) \cap V(J) = L'(x) \cap V(J)$. Since L' satisfies property (C1) with respect to $V(J)$, so does L.

The case when some member of $V(J)$ has an edge in common with a member of $\Sigma_1 \cup \Sigma_2$ is similar but more complicated. The following claim is used. Let D' and D^* be as in Step 4. Then either $D^* \in L(u)$ or $D^* \in L(v)$. The claim is that if $D^* \in L(u)$ (respectively, $L(v)$), then $D^* \in L'(x)$ (respectively, $L'(y)$).

First, suppose that $D^* \in \Sigma_1$. Then $D^* - D \neq \emptyset$ and $D^* \cap D \neq \emptyset$. It follows that D^* has a unique end in H and this end is either u or v. If it is u, then $D^* \in L(u)$ and $D^* - D \subseteq E(H_1)$. Thus, $D^* - D \subseteq E(G_1)$, from which it follows that (G_1, T_1) is an end D-component of (G, T). Moreover, since x is a vertex of G_1 in this case, it is the unique end of D^* in G. Thus, $D^* \in L'(x)$, as required. Similarly, if v is the unique end of D^* in H, then $D^* \in L(v)$, from which is follows that $D^* \in L'(y)$.

Second, suppose that $D^* \in \Sigma_2$. By definition of Σ_2, D^* is a single star, but not a double star, in H. Thus, either u or v is the unique end of D^* in H. Consider the former case. Then $D^* \in L(u)$. Since u is the unique end of D^* in H, D^* is a double star in H_1 and a single star, but not a double star, in H_t. Thus, D^* is a double star in G_1 and a single star, but not a double star, in G_t. It follows that (G_t, T_t) is an end D-component of (G, T) and that y is not an end of D^* in G. This implies that $D^* \in L'(x)$. Similarly, $D^* \in L(v)$ implies $D^* \in L'(y)$. Thus, the claim is proved.

The claim is now used to prove that L satisfies property (C1) with respect to $\Sigma_1 \cup \Sigma_2 \cup V(J)$. Let D' and D^* be as in Step 4. From the claim it follows that if D' is added to $L(u)$ (respectively, $L(v)$) in Step 5, then $D' \in L'(x)$ (respectively, $L'(y)$). Since $I = I(G, \Sigma_3)$, if $D'' \in V(J)$ is added to $L(u)$ (respectively, $L(v)$) in Step 5, then $D'' \in L'(x)$ (respecuvely, $L'(y)$). Since L' satisfies property (C1) with respect to $\Sigma_1 \cup \Sigma_2 \cup V(J)$, the claim implies that L also does. □

The final result of the section is a complexity analysis of Procedures LABEL1, LABEL2, and LABEL3.

LEMMA 3.9. *Procedures* LABEL1, LABEL2, *and* LABEL3 *each requires $O(rc)$ time.*

Proof. Consider Procedure LABEL1. Step 1 requires determining the ends, in H, of each member of Σ_1. Given that the ends of each edge of H are stored with the edge name, determining the ends of $D' \in \Sigma_1$ can be done in $O(|D'|)$ time. Evidently, $|D'| \leq c$. Since Σ_1 is a subset of a decomposer, $|\Sigma_1| \leq r$. Thus, Step 1 requires $O(rc)$ time.

Observe that each vertex of H is in exactly one of $H_1, \dots, H_t$. It follows that for each v that is the end of some member of Σ_1 in H, $L(v) = L_i(v)$ for some $i \in \{1, \dots, t\}$. Thus, Step 2 of the procedure amounts to creating a copy of $L_i(v)$ for each vertex v that is an end of some member of Σ_1. Since each member of Σ_1 has at most two ends, this copying requires $O(|\Sigma_1|)$ time. Thus, Step 2 requires $O(r)$ time.

Consider Procedure LABEL2. The main work is in determining the end, in H, of each member in Σ_2. As in Procedure 1, this requires $O(rc)$ time.

Consider Procedure LABEL3. By the above argument, the initialization in Step 1 requires $O(rc)$ time. Step 2 requires the construction of the graph $I := I(H, \Sigma_3)$. This can be done in $O(rc)$ time as follows. First, construct the union of the members of Σ_3 by scanning each member. Then, for each edge of the union, find the members of Σ_3 that contain it. Each step so far requires $O(rc)$ work. For each pair of members of Σ_3 that contain a given edge of H, an edge of I is constructed. Since there can be at most $O(r^2)$ edges in I, this last step requires $O(r^2)$ work.

Since the graph I has $O(r^2)$ edges, its components can be computed in $O(r^2)$ time. The next step is to determine, for each component J of I, whether some member of $V(J)$ has an edge of H in common with some member of $\Sigma_1 \cup \Sigma_2$. Using a procedure similar to that of the previous paragraph, this can be done in $O(rc)$ time. Thus, the D' and D^* required in Step 4 for each component J can be found in this time. For each component J, the numbers $d(D', D'')$ needed in Step 5 can be computed by breadth-first search applied to J. This requires $O(|E(J)|)$ time for each component J and so requires $O(r^2)$ time overall. □

4. The main algorithm. This section contains the main algorithm, a proof of its correctness, and its time complexity analysis.

Algorithm MAIN.
Input: A standard, connected $r \times c$ $\{0, 1\}$-matrix M.
Output: A gt-realization of M or the conclusion that M is not cr-graphic.
Step 1 : Compute a tree decomposition $\mathcal{T}$ of M, and then for each vertex of $\mathcal{T}$, compute the associated critical decomposer. Let h be the height of $\mathcal{T}$,

and for $0 \leq i \leq h$, compute the set $\Pi_i := \{N \in V(\mathcal{T}) \mid$ the distance from the root of $\mathcal{T}$ to N is $i\}$.

Step 2 : For each $N \in \Pi_h$, construct a consistent gt-realization of N and a corresponding consistent labeling with respect to the critical decomposer of N. Set $i \leftarrow h - 1$.

Step 3 : Choose $N \in \Pi_i$, and let D be the associated minimal d-set. Let $N_1, \ldots, N_t$ be the children of N. Let $(H_1, S_1), \ldots, (H_t, S_t)$ be respective gt-realizations that have been constructed, and, for $1 \leq i \leq t$, let L_i be the labeling of (H_i, S_i) with respect to the critical decomposer of N_i that has been constructed.

Step 4 : If $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ is not mergeable, then stop with the conclusion that M is not cr-graphic. If $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ has more than two end pieces, then stop with the conclusion that M is not cr-graphic. If there does not exist a good merging sequence for $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ with respect to $L_1, \ldots, L_t$ and D, then stop with the conclusion that M is not cr-graphic; otherwise construct such a sequence X, and let (H, S) be the mergence of $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ with respect to D and X.

Step 5 : Compute the principal partition of the critical decomposer of N, and apply Procedures LABEL1, LABEL2, and LABEL3 to compute a labeling L of (H, S) with respect to the critical decomposer of N.

Step 6 : If N is the root of $\mathcal{T}$, then stop with (H, S) as the output. Otherwise, set $\Pi_i \leftarrow \Pi_i - \{N\}$. If $\Pi_i = \emptyset$, then set $i \leftarrow i - 1$. Go to Step 3.

THEOREM 4.1. *Algorithm* MAIN *is correct.*

Proof. Suppose that M is cr-graphic. Each member of Π_h is a matrix with one row. Each such matrix is evidently cr-graphic and has a unique gt-realization that is consistent with respect to the associated critical decomposer. Thus, Step 2 is correct. The gt-realizations and the labelings in Step 3 are evidently well defined when $i = h - 1$, and given that Steps 4–6 are correct, they are well defined for all $i \in \{0, \ldots, h - 2\}$. Moreover, since M is cr-graphic, Step 5 and Lemmas 3.6–3.8 imply that each of the labelings L_i in Step 3 is a consistent labeling. Also since M is cr-graphic, Lemmas 3.2–3.4 imply that Step 4 constructs both a good merging sequence X for $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ and the resulting mergence (H, S). By Lemma 2.7, (H, S) is a gt-realization of N. Thus, in Step 6 when N is the root of $\mathcal{T}$ (i.e., $N = M$), (H, S) is a gt-realization of M, as required.

In the case that M is not cr-graphic, Lemma 2.6 implies that for some N chosen in Step 3, the corresponding set of gt-realizations $\{(H_1, S_1), \ldots, (H_t, S_t)\}$ of its children is not mergeable. Therefore, the algorithms correctly stops in Step 4. □

THEOREM 4.2. *The time complexity of Algorithm* MAIN *is* $O(r^2c)$.

Proof. By Theorem 2.2, the time complexity for computing a decomposition tree $\mathcal{T}$ in Step 1 is $O(rn)$, which is $O(r^2c)$ since $n \leq rc$. The computation of $\Pi_1, \ldots, \Pi_h$ can easily be incorporated into the computation of $\mathcal{T}$ within the same time bound. For future use, it is convenient to also keep track of the set of minimal d-sets associated with the vertices of $\mathcal{T}$ that contain a given column of M. These sets are easily constructed during the computation of $\mathcal{T}$.

The computation of the critical decomposer associated with each vertex of $\mathcal{T}$ can be done as follows. First, for each pair of minimal d-sets (from the set of minimal d-sets associated with the vertices of $\mathcal{T}$), compute the corresponding intersection. This requires $O(c)$ time per pair since each minimal d-set has at most c elements. By Theorem 2.2, $\mathcal{T}$ has $O(r)$ vertices, so finding the collection of all such intersections requires $O(r^2c)$ work. Given the critical decomposer associated with a vertex N of $\mathcal{T}$, the critical decomposer of one of its children N'

in $\mathcal{T}$ can be found by first adding the minimal d-set D associated with N and then by deleting the members that are subsets of D together with the members that are not subsets of $C(N')$. For the child N', deletions of the first and second kind each require $O(rc)$ work; deletions of the first kind can be done using the collection of intersections, and deletions of the second kind can be done by comparing each of the remaining members to $C(N')$. Thus, to find the collection of critical decomposers for the vertex set of $\mathcal{T}$ requires $O(r^2c)$ work.

In Step 2, gt-realizations for the members of Π_h are constructed. There are r such members, one for each row of M. Each such gt-realization has exactly one vertex and at most c edges. Thus, construction of the set of gt-realizations requires $O(rc)$ work. Given the critical decomposer of a particular gt-realization, a consistent labeling can be computed in $O(r)$ time since each critical decomposer has at most r members. Thus, the total work for constructing the set of consistent labelings is $O(r^2)$.

In Step 4, determining whether the given set is mergeable can be done by checking the ends of each edge of D in each of the gt-pairs in the set. Since D has at most c elements and there exist at most r gt-pairs, this requires $O(rc)$ time. The next part of Step 4 requires checking to see if (H_i, S_i) is an end piece, for $1 \leq i \leq t$. Let $\Omega_1, \dots, \Omega_t$ be the respective critical decomposers of $N_1, \dots, N_t$. Observe that for $i \neq j$, $\Omega_i - \{D\}$ is disjoint from $\Omega_j - \{D\}$. It follows that determining which of $(H_1, S_1), \dots, (H_t, S_t)$ are end pieces can be done in $O(rc)$ time by determining the sets $D' \cap D$ for each $D' \in (\Omega_1 - \{D\}) \cup \cdots \cup (\Omega_t - \{D\})$. Given that there exist at most two end pieces, an ordering of $(H_1, S_1), \dots, (H_t, S_t)$ so that the resulting ordered set is good is easily found. The next part of Step 4 is constructing a good merging sequence, if one exists. This can be done by first determining the ends of D in each H_i and then by checking, for each end v of D in H_i, whether $D \in L_i(v)$. Since $|D| \leq c$, finding its ends in a given H_i requires $O(c)$. Since $t \leq r$, this requires $O(rc)$ overall. Scanning the appropriate $L_i(v)$ requires $O(r)$ work since the union of all such $L_i(v)$ has $O(r)$ members. Thus, a good merging sequence, if one exists, can be found in $O(rc)$ time. The final part of Step 4 is the construction of (H, S). This can be done by deleting D from each of $(H_1, S_1), \dots, (H_t, S_t)$, adding edges joining consecutive vertices in the merging sequence, and then adding the edges of D with the appropriate ends. Since $|D| \leq c$ and $t \leq r$, this step requires $O(rc)$ work.

Step 5 first computes the principal partition of the critical decomposer Ω of N. This can be done by first determining, for each member of Ω, whether it is in the critical decomposer of one of the children of N. This requires $O(r^2)$ work since $|\Omega| \leq r$ and the cardinality of the union of the critical decomposers of the children of N is $O(r)$. For a member of Ω that is not in one of the critical decomposers of the children of N, it needs to be determined whether it is a double star of H. For a given member, this can be done in $O(c)$ time by finding the ends of each edge in the member. Since Ω has at most r members, the total work for this part is $O(rc)$, so the principal partition can be found in $O(rc)$. Step 5 then applies Procedures LABEL1, LABEL2, and LABEL3, which is $O(rc)$ by Lemma 3.9.

Step 6 ensures that each of Steps 3–5 is executed at most $O(r)$ times, once for each vertex of $\mathcal{T}$. Thus, Algorithm MAIN requires $O(r^2c)$ time. □

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] R. E. Bixby and D. K. Wagner, *An almost-linear time algorithm for graph-realization*, Math. Oper. Res., 13 (1988), pp. 99–123.

[3] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, New York, 1976.

[4] K. S. Booth and G. S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[5] P. DIETZ, M. FURST AND J. E. HOPCROFT, *A linear-time algorithm for the generalized consecutive retrieval problem*, Tech. report TR–79–386, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

[6] S. FUJISHIGE, *An efficient PQ-graph algorithm for solving the graph realization problem*, J. Comput. System Sci., 21 (1980), pp. 63–86.

[7] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices with the consecutive* 1*'s property*, Bull. Amer. Math. Soc., 70 (1964), pp. 681-684.

[8] ———, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[9] S. P. GHOSH, *File organization: the consecutive retrieval property*, Comm. ACM, 15 (1972), pp. 802–808.

[10] ———, *On the theory of consecutive storage of relevant records*, Inform. Sci., 6 (1973), pp. 1–9.

[11] ———, *File organization: consecutive storage of relevant records on a drum-type storage*, Inform. and Control (Shenyang), 25 (1974), pp. 145–165.

[12] ———, *Consecutive storage of relevant records with redundancy*, Comm. ACM, 18 (1975), pp. 464–471.

[13] U. GUPTA, *Bounds on storage for consecutive retrieval*, J. ACM, 26 (1979), pp. 28–36.

[14] L. T. KOU, *Polynomial complete consecutive information retrieval problems*, SIAM J. Comput., 6 (1977), pp. 67–75.

[15] W. LIPSKI, *Information storage and retrieval–mathematical foundations* II (*combinatorial problems*), Theoret. Comput. Sci., 3 (1976), pp. 183–211.

[16] T. NAKANO, *A characterization of intervals; the consecutive* (*one's or retrieval*) *property*, Comment. Math. Univ. St. Paul., 22 (1973), pp. 49–59.

[17] ———, *A remark on the consecutivity of incidence matrices*, Comment. Math. Univ. St. Paul., 22 (1973), pp. 61–62.

[18] A. N. PATRINOS AND S. L. HAKIMI, *File organization with consecutive retrieval and related properties*, in Large Scale Dynamical Systems, R. Sacks, ed., Point Lobos Press, North Hollywood, CA, 1974, pp. 276–297.

[19] R. SWAMINATHAN AND D. K. WAGNER, *The arborescence-realization problem*, Discrete Appl. Math., to appear.

[20] K. TANAKA, *Tree-structured data organization with consecutive retrieval property*, in Polish Conference on Consecutive Retrieval, W. Lipski and S. Ghosh, eds., Academic Press, New York, 1983, pp. 271–276.

[21] M. TRUSZCZYŃSKI, *An algorithm of finding an acyclic f-graph for a family of sets*, Fund. Inform., 3 (1980), pp. 379–396.

[22] A. C. TUCKER, *Characterizing the consecutive* 1*'s property*, in Proceedings of the Second Chapel Hill Conference on Combinatorial Mathematics and its Applications, University of North Carolina, Chapel Hill, NC, 1970, pp. 472–477.

[23] ———, *A structure theorem for the consecutive* 1*'s property*, J. Combin. Theory, 12 Ser. B (1972), pp. 153–162.

[24] W. T. TUTTE, *An algorithm for determining whether a given binary matroid is graphic*, Proc. Amer. Math. Soc., 11 (1960), pp. 905–917.

[25] A. WAKSMAN AND M. W. GREEN, *On the consecutive retrieval property in file organization* IEEE Trans. Comput., C–23 (1974), pp. 173–174.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 415–436, April 1994

POLYNOMIAL ROOT-FINDING ALGORITHMS AND BRANCHED COVERS*

MYONG-HI KIM† AND SCOTT SUTHERLAND†

Abstract. A family of root-finding algorithms is constructed that combines knowledge of the branched covering structure of a polynomial with a path-lifting algorithm for finding individual roots. In particular, the family includes an algorithm that computes an ϵ-factorization of a polynomial of degree d that has an arithmetic complexity of $\mathcal{O}\big(d(\log d)^2|\log\epsilon| + d^2(\log d)^2\big)$. At the present time, this complexity is the best known in terms of the degree.

Key words. Newton's method, approximate zeros, arithmetic complexity, path-lifting method, branched covering

AMS subject classifications. primary 68Q25; secondary 58C10, 65H05, 30C15, 58F08

Introduction. The problem of devising optimal methods for numerically approximating the roots of a polynomial has been of interest for several centuries, and is far from solved. There are numerous recent works on root-finding algorithms and their cost, for example, the work of Jenkins and Traub [JT70], Renegar [Ren87], Schönhage [Sch82], and Shub and Smale [SS85], [SS86], [Sma85]. This list is far from complete; the reader should refer to the aforementioned papers as well as [DH69] for more detailed background. The work in this paper is most closely related to that of Smale.

Our algorithm computes an approximate factorization of a given polynomial (that is, it approximates all the roots). In constructing it, we combine global topological information about polynomials (namely, that they act as branched covers of the Riemann Sphere) with a path-lifting method for finding individual roots. Utilizing this global information enables us to use fewer operations than applying the path-lifting method to each root sequentially.

Renegar's algorithm in [Ren87] approximates all d roots of a given polynomial using $\mathcal{O}\big(d^3\log d + d^2(\log d)(\log|\log\epsilon|)\big)$ arithmetic operations in the worst case. He has shown that the factor of $\log|\log\epsilon|$ in the complexity is the best possible if one restricts to the operations $+$, $-$, $\times$, and $\div$. This algorithm has a component (the Shur-Cohn algorithm) that requires exact computation and so is not suitable for an analysis of bit complexity, that is, one that accounts for rounding errors introduced by finite precision. In [Pan87], Pan gives a different algorithm that slightly improves the complexity to $\mathcal{O}\big(d^2\log d\log|\log\epsilon|\big)$; this algorithm also operates effectively as a parallel algorithm.

Schönhage [Sch82] gives an algorithm that produces an ϵ-factorization with a bit complexity of $\mathcal{O}\big(d^3\log d + d^2|\log\epsilon|\big)\log(d|\log\epsilon|)\log\log(d|\log\epsilon|)$, via the "splitting circle method." Note that the customary parameter for bit length of the coefficients does not appear in the complexity. This is because, as Schönhage states, for fixed degree d and output precision ϵ, there is a number s_0 for which "the input [coefficients] a_ν can be restricted to complex integer multiples of 2^{-s_0} without loss of generality." In [Ren87], it is stated that Schönhage believes that, if exact arithmetic is used, this method "should yield a complexity bound [in ϵ] of $\mathcal{O}(d^\alpha\log|\log\epsilon|)$, most likely with $\alpha \le 3$."

Smale's path lifting algorithm in [Sma85] approximates a single root of the polynomial with a worst case arithmetic complexity of $\mathcal{O}\big(d(\log d)|\log\epsilon| + d^2(\log d)^2\big)$, and an average complexity of $\mathcal{O}\big(d^2 + d|\log\epsilon|\big)$. One good feature of this line of work is that it is stable under round-off error. In [Kim89a], Kim shows that if f and f' are computed with relative error 10^{-3} until an approximate zero (see §1.2) is reached, then the algorithm behaves exactly the same.

*Received by the editors July 8, 1991; accepted for publication March 25, 1993. Part of this work was done while the first author was at Bellcore, Morristown, New Jersey. An earlier version of this paper was circulated with the title "Parallel Families of Polynomial Root-Finding Algorithms."

†Institute for Mathematical Sciences, State University of New York at Stony Brook, Stony Brook, New York 11794-3600 (myonghi@math.sunysb.edu and scott@math.sunysb.edu).

A recent series of papers by Shub and Smale [SS93a], [SS93b], [SS93c], [SS93d] generalizes the path lifting algorithm to systems of homogeneous polynomials in several variables.

The algorithm presented here exploits the branched covering structure of a polynomial to choose good starting points for a variant of Smale's algorithm, and we obtain a worst-case arithmetic complexity of $\mathcal{O}\big(d(\log d)|\log\epsilon| + d^2(\log d)^2\big)$ to compute an ϵ-factorization. In a subsequent paper, we shall compute the bit complexity of this algorithm. Because of the stability mentioned in the previous paragraph and the ability to exploit bounds on the variation of f and f', we hope to achieve results comparable to Schönhage's.

At first glance, it may appear that our complexity results are inferior to some of those above in terms of ϵ. However, in practice there is usually a relationship between the degree d and the desired precision ϵ. In particular, if we have $\epsilon \geq 2^{-d}$, then the complexity of our algorithm compares favorably with all of those mentioned above. Furthermore, our algorithm is simple to implement and is numerically very stable.

Our algorithm is suitable for some amount of parallelization, but has a sequential component of $\mathcal{O}(d + |\log\epsilon|)$ operations. However, we think of this algorithm as acting on d points simultaneously, and techniques that evaluate a polynomial at d points (see [BM75], for example) are used to cut the cost involved. Of course, the algorithm can be implemented on a sequential machine while still taking advantage of these techniques. In fact, evaluation of the polynomial is the only point at which we need to use asymptotic estimates to achieve the stated complexity; the other places where we use asymptotic estimates are only for ease of expositon.

The reader should also see the papers [BFKT88], [BT90], [Nef90] for fully parallel algorithms for solving polynomials with integer coefficients. In [BFKT88], it is shown that if all roots of the polynomial are real, this problem is in NC. Neff extends this result to allow complex roots in [Nef90].

This paper is structured as follows. In §1, after some background material, we recall the "path lifting method" of Smale and present a version of the relevant theorem (our Theorem 1.5) that improves the constants involved somewhat. We then discuss how we can exploit the branched-covering structure of a polynomial to choose initial points for the algorithm, many of which will converge to roots. We close the section with a brief explanation of how to construct families of algorithms that locate $\frac{d}{n}$ roots at a time, for various values of n.

Section 2 presents an explicit algorithm for a specific family, which locates $\frac{d}{2}$ points at a time. Our main theorem, Theorem 2.1, states that this algorithm always terminates with an ϵ-factorization of the input polynomial, and gives a bound on the number of arithmetic operations required in the worst case. As a corollary, the algorithm can be used to locate all d roots of the polynomial to within ϵ with a complexity of $\mathcal{O}\big(d^2(\log d)|\log\epsilon| + d^2(\log d)^2\big)$. In the subsequent sections, each component of the algorithm is analyzed, and the relevant lemmas are proven. Finally, we tie all the components together and prove the main theorem.

1. Preliminaries.

1.1. Root and coefficient bounds. Given a polynomial $\phi(z) = \sum_{i=0}^{d} a_i z^i$, with $a_i \in \mathbb{C}$, it is our goal to determine an approximate factorization of ϕ, that is, approximations $\hat{\xi}_i$ to the actual roots ξ_i of ϕ so that $\|\phi(z) - \prod(z - \hat{\xi}_i)\| < \epsilon$. The norm we shall use here is the max-norm, that is, $\|\phi\| = \max|a_j|$. A related problem is to ensure that $|\xi_i - \hat{\xi}_i| < \epsilon'$; there are well-known estimates giving the relationship between ϵ and ϵ', so solving one problem essentially solves the other.

In order to have an estimate on the complexity of a root-finding algorithm, we need a compactness condition on the space of polynomials. This can be done either by placing conditions of the location of the roots or on the coefficients; such bounds are interrelated.

Since our goal is to minimize a functional norm, it seems most natural to place our conditions on the coefficients. We shall assume our input polynomial ϕ is an element of the family

$$\mathcal{P}_d(1) = \left\{ z^d + \sum_{j=0}^{d-1} a_j z^j, \text{ with } |a_j| \le 1 \right\}.$$

This is the same space as used by Smale and others [SS85], [Sma81], [Sma85]. One can always transform an arbitrary polynomial into an element of $\mathcal{P}_d(1)$: if $p(z) = \sum_{i=0}^{d} b_i z^i$, and $B = \max |b_j/b_d|^{1/(d-j)}$, then $p(Bz)/B^d \in \mathcal{P}_d(1)$.

One should not confuse this family with the degree d polynomials whose roots are in the unit disk, although unfortunately this space is also often denoted by $\mathcal{P}_d(1)$ (for example, in [Fri90] and [Ren87]).

There are a number of estimates that relate the coefficients of a polynomial to a bound on the modulus of the zeros (see [Hen74] or [Mar66], for example). Such an estimate is important to us, since although membership in $\mathcal{P}_d(1)$ is not preserved under deflation (division of factors), bounds on the modulus of the roots are. We state one such bound here (Corollary 6.4k of [Hen74]).

LEMMA 1.1. *All the zeros of the polynomial $z^d + \sum_{j=0}^{d-1} a_j z^j$ lie within the open disk with center* 0 *and radius*

$$2 \max_{0 \le j < d} |a_j|^{1/(d-j)}.$$

As an immediate consequence, we see that the roots of a polynomial in $\mathcal{P}_d(1)$ lie within $\mathbb{D}_2$.

1.2. Approximate zeros. Our algorithm uses a path lifting method (see below) to get close to the roots of our polynomial, and then uses the standard Newton's method to further refine these approximations. This is done because Newton's method converges very quickly in a neighborhood of a simple root, but can fail for some initial points outside this neighborhood. One of the authors [Sut89] has shown how one can guarantee convergence of Newton's method, but a bound on the arithmetic complexity has not been computed. Instead, we use the more certain path lifting method as described in §1.3; this allows an explicit computation of the complexity.

Following Smale [Sma81], we call a point z_0 an *approximate zero* if Newton's method converges rapidly (that is, quadratically) when started from z_0. Such terminology is reasonable, because given such a point, we can quickly obtain an approximation of a root to arbitrary precision.

DEFINITION 1.2. *Let f be a polynomial and let z_n be the nth iterate under Newton's method of the point z_0, that is, $z_n = z_{n-1} - f(z_{n-1})/f'(z_{n-1})$. Then we say that z_0 is an approximate zero of f if, for all $n > 0$ we have*

$$|z_n - \zeta| \le 8 \left(\frac{1}{2}\right)^{2^n} |z_0 - \zeta|,$$

for some root ζ of f.

Notice that this definition is never satisfied in the neighborhood of a multiple root of f, since the convergence of Newton's method is asymptotically linear there. In our algorithm, we perturb the polynomial slightly to ensure that we always have simple zeros. Refer to §§2.1 and 2.3 for more details.

Kim [Kim88] and Smale [Sma86] have developed readily tested criteria for determining, based on the values of the derivatives $f^{(k)}(z)$, when a point z is an approximate zero. These can be extended to a much more general setting, namely for f a mapping between Banach spaces. The following is essentially Theorem A of [Sma86].

LEMMA 1.3. *Let*

$$\alpha_f(z) = \max_{k>1} \left|\frac{f(z)}{f'(z)}\right| \left|\frac{f^{(k)}(z)}{k!\,f'(z)}\right|^{1/(k-1)}.$$

If $\alpha_f(z) < \frac{1}{8}$, *then* z *is an approximate zero of* f.

We will find the following also very useful.

LEMMA 1.4. *Let* $f(z)$ *be a polynomial and* z *be a complex number so that* $f'(z) \neq 0$, *and let* $R_f(z)$ *be the radius of convergence of the branch of the inverse* f_z^{-1} *which takes* $f(z)$ *to* z. *If*

$$\frac{|f(z)|}{R_f(z)} < \frac{1}{10},$$

then z *is an approximate zero of* f. *Furthermore, if we have* $|f(z)|/R_f(z) < \frac{1}{32}$, *then* $\alpha_f(z) < \frac{1}{8}$.

Remark. If Smale's mean value conjecture holds (see [Sma81], [Tis89]), then the hypotheses of the lemma imply that $\alpha_f(z) < \frac{1}{8}$.

Proof. The first result is a consequence of the proof of Theorem 4.4 of [Kim88], and the second is an immediate consequence of Corollary 4.3 of the same paper. This, in turn, uses the Extended Loewner's Theorem in [Sma81]. □

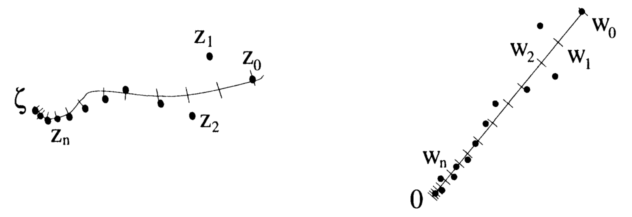
1.3. The path lifting method. Here we review the path lifting method, which forms the core of our iteration scheme. This method is sometimes referred to as a "generalized Euler method" or "modified Newton's method"; we prefer the term "path lifting method" as it is the most descriptive (to us, anyway). This method appears in the work of Smale [Sma85], although the version we present here is slightly different and we present another proof of the relevant theorem, which is quite simple. It should be emphasized that the path lifting method, like Newton iteration, is an algorithm for finding a *single* root of a polynomial; we discuss how to combine these to find all roots in §1.5 below.

We think of a polynomial f as a map from the *source space* to the *target space*; that is, $f : \mathbb{C}_{\text{source}} \to \mathbb{C}_{\text{target}}$. Given an initial value z_0 in the source space, we connect its image $w_0 = f(z_0)$ to 0 in the target space, and then lift this ray under the proper branch of f^{-1} to a path connecting z_0 with a root ζ of f. Of course, we don't explicitly know this inverse, but if the path in the target space stays well away from the critical values of f, the local inverse map $f_{z_0}^{-1}$ is well-defined on a neighborhood of the ray. Even if the path *does* contain critical values, a local inverse can still be defined for some z_0. See §1.4.

The basic idea of the path lifting method is to take a sequence of points w_n along the ray in the target space, with $w_0 = f(z_0)$. We then construct a sequence of points z_n in the source space so that $f(z_n)$ is near w_n in the target. See Fig. 1.1. This is done using a single step of Newton's method to solve $f(z) = w_n$ with z_{n-1} as the starting point. That is,

$$z_n = z_{n-1} - \frac{f(z_{n-1}) - w_{n-1}}{f'(z_{n-1})}.$$

This construction will converge to a root ζ in the source space if there is a wedge about the ray in the target space on which there is a well-defined branch of the inverse f_ζ^{-1}, and if the w_n are chosen properly (in a way that depends on the angle of the wedge). The larger the wedge about the ray, the faster the method converges. We now state the exact theorem, although we shall defer the proof until §2.3.



source space

target space

FIG. 1.1. *The source and target spaces in the path lifting method. In the source space, each z_i is indicated by a black dot, and $f(z_i)$ is indicated by a black dot in the target space. Similarly, the w_i are indicated by tick marks in the target space, and $f^{-1}(w_i)$ by ticks in the source space.*

Notation. By a wedge $\mathcal{W}_{A,w}$, we mean the set

$$\{z \mid 0 < |z| < 2|w|, \ \arg w - A < \arg z < \arg w + A\}.$$

THEOREM 1.5. *Suppose that the branch of the inverse $f_{z_0}^{-1}$ is analytic on a wedge $\mathcal{W}_{A,w_0}$, with $0 < A \le \frac{\pi}{2}$, and let $h \le \frac{\sin A}{19}$. Suppose also that $|f(z_0) - w_0| < h|w_0|/2$, and define*

$$w_n = (1-h)^n w_0, \qquad z_{n+1} = z_n - \frac{f(z_n) - w_{n+1}}{f'(z_n)}.$$

Then $|f(z_n) - w_n| \le h|w_n|/2$ and $z_{n+1} \in f_{z_0}^{-1}\left(\mathcal{W}_{A,w_n}\right)$.

It should be noted that this theorem is a slight improvement of Smale's Theorem A in [Sma85]. His proof is valid for all angles, but is stated only for $A = \frac{\pi}{12}$ with $h = \frac{1}{98}$, and for $A = \frac{\pi}{4}$ with $h = \frac{1}{32}$. For $A = \frac{\pi}{12}$ we can take take $h = \frac{1}{74}$, and for $A = \frac{\pi}{4}$, $h = \frac{1}{27}$ is adequate.

1.4. Branched covers, inverse functions, and all that. If f is a polynomial of degree d, then $f : \mathbb{C} \to \mathbb{C}$ is a branched covering with branch points at the critical points θ_i of f. If z is a regular point of f, that is, $f'(z) \ne 0$, then there is a well-defined inverse function f_z^{-1} so that $f_z^{-1}(f(z)) = z$.

In any neighborhood of a critical point of f, there cannot be a single valued inverse; however, the behavior at such points is well understood. Let θ be a critical point of multiplicity $k-1$. Then we have

$$f(z) - f(\theta) = (z-\theta)^k g(z), \qquad \text{where} \quad g(\theta) \ne 0.$$

One can then define k branches of the inverse that are analytic on a small slit disk about $f(\theta)$. We may, of course, choose any slit that connects $f(\theta)$ to the boundary of the disk. The reader is referred to a complex analysis text for further details (for example, see [Ahl79]).

Let $\{\zeta_j\}$ be the d roots of f, represented with multiplicity. If ζ_j is a simple root, denote by $f_{\zeta_j}^{-1}$ or f_j^{-1} the unique branch of the inverse of f that takes 0 to ζ_j. On the other hand, if ζ_j is a multiple root, we let $f_{\zeta_j}^{-1} = f_j^{-1}$ be one of the branches of the inverse that take 0 to ζ_j, taking care to account for all such branches exactly once. This can be done, since if ζ is a root of multiplicity $k \ge 2$, it is also a critical point of multiplicity $k-1$, and so there are k branches of the inverse.

We now analytically continue each of the f_j^{-1} to a maximal starlike domain Ω_j in the target space; that is, we attempt to extend each f_j^{-1} along open rays from 0. When doing this, it is useful to think of the target space as consisting of d copies of $\mathbb{C}$, with a single f_j^{-1} associated with each one. When does the analytic continuation fail? Precisely when the inverse image of a ray encounters a critical point of f. Refer to Fig. 1.2.

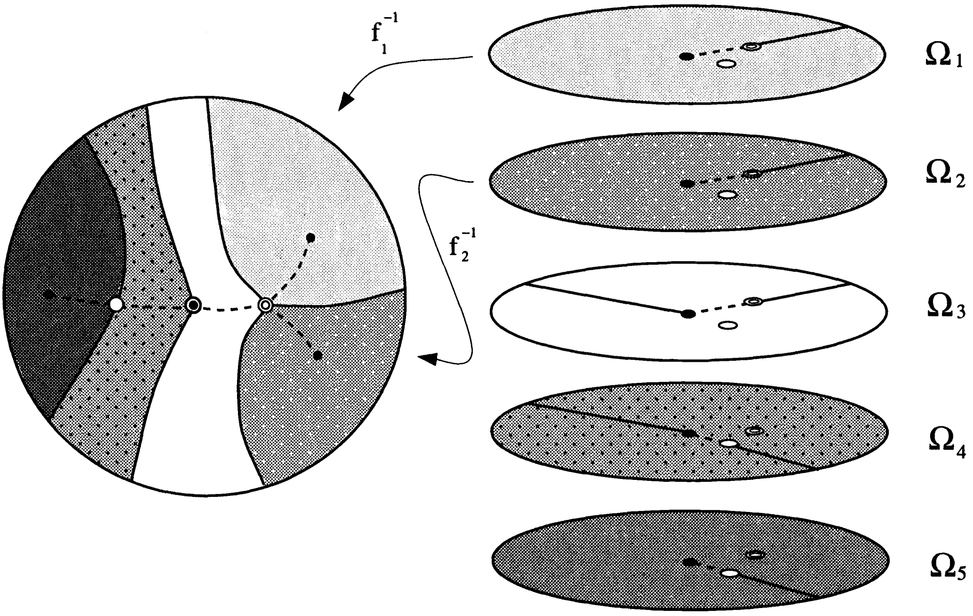


FIG. 1.2. *The Ω_j in the target space (on the right), and the corresponding source space, for a degree 5 polynomial with a double root (black dot inside a white dot) and a critical point of multiplicity 2 (double white dot). The other critical point is marked by a single white dot, and the roots by black dots. The cuts in the Ω_j are represented by black radial lines.*

At this point, it may be useful to consider the Newton vector field given by

$$\dot{z} = -f(z)/f'(z).$$

Let $\varphi_t(z)$ be the solution curve with initial condition $\varphi_0(z) = z$. Then we have

$$f(\varphi_t(z)) = e^{-t} f(z);$$

that is, f maps solution curves of the Newton vector field to rays in the target space. Notice that the singularities of the vector field occur precisely at the points where $f'(z) = 0$. The solution curves $\varphi_t(z)$ which have singularities play an important role here: they divide the source space into regions on which f is injective. Refer to [Shu83], [STW88], [Sma85] for more details on the behavior of the solution curves. Applying the path lifting method can be viewed as attempting to follow the solution curves to the flow φ_t.

When constructing the Ω_j, we continue f_j^{-1} radially outward until a critical point θ is encountered in the source space. We then exclude the ray $\{rf(\theta) \mid r \geq 1\}$ from Ω_j, and continue by moving along rays that avoid the cut. Notice that when we encounter a critical point θ of multiplicity $k-1$, we need to slit at most k of the Ω_j starting at $f(\theta)$. Also, note that some of the Ω_j may already be slit at $f(\theta)$, since there may be another critical point whose image lies on the same ray.

We now count the number of such cuts: f has $d-1$ critical points (with multiplicity), and a critical point of multiplicity $k-1$ can cause at most k cuts. This means we have at most

$2(d-1)$ cuts, distributed through the d copies of the target space. Note that if Ω_j contains some wedge W, then f_j^{-1} is analytic on W. The following counts the number of Ω_j which contain wedges of a given size.

LEMMA 1.6. *Let m be an integer, and divide $\mathbb{C}$ into m wedges*

$$W_{n,m} = \left\{ w \;\middle|\; \frac{2n\pi}{m} < \arg w < \frac{2(n+1)\pi}{m} \right\} \qquad n = 0, \dots, m-1.$$

For each wedge $W_{n,m}$, let $N(n,m)$ be the number of Ω_j that contain the sector $W_{n,m}$, and let $N(m) = \max_{0 \le n < m} N(n,m)$. Then

$$N(m) \ge d - \left\lfloor \frac{2(d-1)}{m} \right\rfloor.$$

Proof. Since we have $2(d-1)$ cuts and m wedges W_n, at least one of the wedges has no more than $\frac{2(d-1)}{m}$ cuts. Since there are d Ω_js, we have the result. □

COROLLARY 1.7. *$N(d) = d$, $N(3) \ge \frac{d}{3}$, and $N(4) \ge \frac{d}{2}$.*

Proof. Application of the formula above gives the result for $N(3)$ and $N(4)$, and yields $N(d) \ge d-1$. However, since each critical point causes at least two cuts, it is not possible to have a wedge cut only once. Thus, the value $d-1$ is not permissible for $N(d)$, giving $N(d) = d$. □

1.5. Families of root-finding methods. We now use the result of Lemma 1.6 to construct families of root-finding algorithms. Recall that the modified Newton method described in §1.3 works when there is an Ω_j containing a wedge about our initial value $f(z_0)$; the larger the wedge, the faster the method converges.

For each family, we start with md points in the source space placed around a circle that contains all the roots. We think of this as m sets of d initial points, and choose them so that the image of each set lies well inside each of the m sectors $W_{n,m}$ in the target space. Then by Lemma 1.6, one of the m sets of points will contain at least $N(m)$ elements whose images are each in a "good wedge," that is, they lie in some Ω_j. As a consequence, iterating these points under the path lifting method will locate at least $N(m)$ roots of the polynomial.

Particular families of interest are $m = d$, which gives the algorithm discussed in [Kim89b], and $m = 4$, on which we focus our attention here. The basic idea of all of the algorithms is this: obtain md "good" initial points and apply the path lifting method to d of them at a time. If, after a prescribed number of iterations, we have found approximation to at least $N(m)$ roots (counting multiplicity), we deflate the polynomial (that is, divide out the approximated roots) and repeat the process on the result. If not, we try again with the next set of d points. Note that we are guaranteed success by the time we try the mth set.

The remainder of the paper consists of a detailed description and analysis of the algorithm for $m = 4$. Most of what follows can be readily adapted to the other families as well.

2. A root-finding algorithm.

2.1. Statement of the algorithm and main theorem. Here we present our root-finding algorithm for the family $m = 4$. The presentation is structured as a main routine and several subroutines, which do most of the work.

Notation. Throughout this section, we shall denote matrices, vectors, and sets in uppercase calligraphic type, and their elements in subscripted lowercase type. For example, x_j is the jth element of the vector $\mathcal{X}$. We shall also use the notation $\lfloor x \rfloor$ to denote the least integer in x, sometimes also called `floor(x)`.

The main routine merely inputs the desired polynomial and precision, rescales it so the roots lie in the disk of radius $\frac{1}{2}$, then repeatedly calls a subroutine to halve the number of unknown roots (counted with multiplicity) and deflate. We do the rescaling in order to easily bound the error introduced by the FFT deflation. The set Λ contains all the approximations found by the ith stage.

Note that the algorithm is given for an arbitrary monic polynomial, since only minor changes are required to normalize the input polynomial. If it is assumed that the input polynomial is already in $\mathcal{P}_d(1)$, we can take $f_0(z) = \phi(4z)/4^d$ and $\tau = 32\epsilon/7^{d+3}$.

```
Main Routine

Input monic polynomial $\phi(z) = \sum_{i=0}^{d} a_i z^i$ and desired precision $\epsilon$.
Let $f_0(z) = \phi(Kz)/K^d$, with $K = 4 \max |a_j|^{\frac{1}{d-j}}$.
Let $\tau = \frac{\epsilon}{2K^d} (4/7)^{d+3}$.
Let $i = 0$ and $\Lambda = \emptyset$.
While $\#(\Lambda) < \deg(\phi)$
    Let $(f_{i+1}, \Lambda_i) =$ get-half-roots-and-deflate($f_i, \tau$).
    Let $\Lambda = \Lambda \cup \Lambda_i$.
    Increment $i$.
End While.
Output $K\Lambda$.
```

The function get-half-roots-and-deflate takes as input a normalized polynomial f and precision τ. It returns a set of points y_j that approximate at least half of the roots of f (with multiplicity) and a new polynomial $\tilde{f}$ that we obtain by deflation. These satisfy $\| f(z) - \tilde{f}(z) \prod (z - y_j) \| < 2\tau$. We should point out here that we are actually finding approximate zeros of $f - \vec{\tau}$, where $|\vec{\tau}| = \tau$, which depends on ϵ. When the translation is in the proper quadrant, this will ensure that the relevant roots of $f - \vec{\tau}$ are simple, so that we have approximate zeros in a neighborhood. This allows us to obtain the right number of approximations to a multiple root, without worrying about winding number arguments or the like. We emphasize again that τ is chosen as a function of ϵ, and is small enough that the approximation polynomial has negligible errors in the non-constant terms.

The matrix $\mathcal{Z}$ consists of 4 rows of d "good" initial conditions, with $|z_{j,k}| = \frac{3}{2}$ and $\arg f(z_{j,k}) \approx 2\pi i j/4$. We use $\mathcal{Z}_j$ to represent the jth row.

```
function get-half-roots-and-deflate($f, \tau$)

Let $d = \deg f$.
Let $\mathcal{Z}$ =choose-4d-good-initial-points($f$).
For $j = 1$ to 4 do
    Let $\mathcal{Y} =$ iterate-PLM($f, \mathcal{Z}_j, j, \tau$).
    Let $\psi = f - \tau e^{2\pi i j/4}$.
    Let $\mathcal{X} =$ select-approx-zeros($\psi, \mathcal{Y}$).
    Let $\mathcal{W} =$ polish-roots($\psi, \mathcal{X}, \tau$).
    Let $\mathcal{V} =$ weed-out-duplicates($\psi, \mathcal{W}$).
    If $\#(\mathcal{V}) \geq d/2$ then
        Let $\tilde{f} =$ deflate($\psi, \mathcal{V}$).
        Return $(\tilde{f}, \mathcal{V})$.
    End if.
End for.
```

```
Print "We have proven this statement will not be reached. Check your code."
Abort.
```

The function `choose-4d-good-initial-points` gives us our 4 sets of d initial values on the circle of radius $\frac{3}{2}$. The sets have the property that elements of the same set are mapped to points with approximately the same argument, and elements of distinct sets are mapped to points in different quadrants of the target space. There are several different ways to accomplish this, and other methods can slightly decrease the number of operations required in `iterate-PLM`. Please refer to the remarks in §2.2 for more details.

```
function choose-4d-good-initial-points($f$)

Let $N = 676 \deg f$.
For $k = 1$ to $N$
    Let $\omega_k = \frac{3}{2}e^{2\pi ik/N}$.
End For.
For $j = 1$ to 4
    Let $\mathcal{Z}_j$ be the union of the $\omega_k$ for which $\arg f(\omega_k) \leq 2\pi/j$ and $\arg f(\omega_{k+1}) > 2\pi/j$.
End For.
Return($\mathcal{Z}$).
```

From our sets of initial points $\mathcal{Z}_j$, we obtain our approximate zeros via the routine `iterate-PLM`. This applies the path lifting method to f for an appropriate number of steps. For simplicity, we present a scalar version here. However, there are well-known methods for evaluating the same polynomial f at m different points ($m \geq d$) in $\mathcal{O}\big(m(\log d)^2\big)$ arithmetic operations; refer to §4.5 of [BM75]. When computing the complexity in §2.7, we assume such methods are used. Note that this algorithm can be easily implemented on either a vector or parallel computer, so that one can iterate the d elements of $\mathcal{Z}_j$ simultaneously. Also note that if Smale's mean value conjecture holds, the extra iteration to obtain $\hat{z}$ is not needed; z_N is good enough. See the remarks following Lemma 1.4.

```
function scalar-iterate-PLM($f, z_0, j, \tau$)

Let $w_0 = |f(z_0)|e^{j\pi i/2}$.
Let $h = 1/27$.
Let $N = \left\lfloor \dfrac{\log_2(\tau/|w_0|)}{\log_2(26/27)} \right\rfloor$.
For $n = 1$ to $N$
    Let $w_n = (1-h)w_{n-1}$.
    Let $z_n = z_{n-1} - \dfrac{f(z_{n-1}) - w_i}{f'(z_{n-1})}$.
End for.
Let $\hat{z} = z_N - \dfrac{f(z_N) - \tau e^{j\pi i/2}}{f'(z_N)}$.
Return($\hat{z}$).
```

The next routine takes the output of `iterate-PLM` and uses the α function (defined in Lemma 1.3) to remove those elements that are not approximate zeros. Although the test $\alpha < \frac{1}{8}$ is sufficient, it is not a necessary condition. However, if the image of the initial points $\mathcal{Z}_j$ lie in a "good quadrant," we know by Lemma 2.5 below that we will have $\alpha < \frac{1}{8}$ for at least half of them, and so we are not in danger of discarding too many points.

```
function select-approx-zeros($\psi, \mathcal{Y}$)

Let $\mathcal{X} = \emptyset$.
For $j = 1$ to $\#(\mathcal{Y})$
    If $\alpha_\psi(y_j) < \frac{1}{8}$, then let $\mathcal{X} = \mathcal{X} \cup \{y_j\}$.
End For.
Return($\mathcal{X}$).
```

Once we have found approximate zeros for at least half of the roots of f_i (by applying `iterate-PLM` to at most 4 sets $\mathcal{Z}_j$), we further refine them by "polishing" with regular Newton's method. As in `iterate-PLM`, fast polynomial evaluation techniques can be used, but we present a scalar version here for simplicity.

```
function scalar-polish-roots($\psi, x_0, \tau$)

Let $M = 1 + \lfloor \log_2 \log_2 \left(64d(7/4)^d\right) / \tau \rfloor$.
For $n = 1$ to $M$
    Let $x_n = x_{n-1} - \dfrac{\psi(x_{n-1})}{\psi'(x_{n-1})}$.
End For.
Return($x_M$).
```

Finally, we remove from the set of approximations any points that approximate the same root of ψ. We do this by taking the approximation that minimizes $|\psi|$, and then making a pass through the rest of them and accepting only those that approximate different roots from the ones previously accepted.

```
function weed-out-duplicates($\psi, \mathcal{W}$)

Sort $\mathcal{W}$ so that $|\psi(w_1)| \leq |\psi(w_2)| \leq \ldots \leq |\psi(w_n)|$.
Let $\mathcal{V} = \{w_1\}$.
For $j = 2$ to $\#(\mathcal{W})$
    If $|w_j - v| > 3|\psi(w_j)|/|\psi'(w_j)|$ for all $v \in \mathcal{V}$, then let $\mathcal{V} = \mathcal{V} \cup \{w_j\}$.
End For.
Return($\mathcal{V}$).
```

At this point we have found approximations to at least half of the roots of f_i. We divide them out to obtain a new polynomial f_{i+1} of smaller degree, using a standard technique involving the finite Fourier matrix.

```
function deflate($\psi, \mathcal{V}$)

Let $n = \deg \psi - \#\mathcal{V}$.
Let $\omega = e^{2\pi i/(n+1)}$.
Let $p(z) = \prod_{k=1}^{\#\mathcal{V}} (z - v_k)$.
Let $\mathcal{M}^{-1}$ be the inverse of the $n \times n$ Fourier matrix. That is, $m_{j,k} = \omega^{-jk}/n$.
Let $\mathcal{P}$ be the column vector with entries $p_j = \dfrac{\psi(\omega^j)}{p(\omega^j)}$, $j = 0, \ldots, n$.
Let $\mathcal{Q} = \mathcal{M}^{-1}\,\mathcal{P}$.
Let $q(z) = \sum_{j=0}^{n} q_j z^j$.
Return($q$).
```

We now state our main theorem, which essentially says that the algorithm just presented works.

THEOREM 2.1. *Given a monic polynomial $\phi(z)$ of degree d and $\epsilon > 0$, the algorithm presented in this section will always terminate with d points $\lambda_1, \dots, \lambda_d$ that satisfy*

$$\|\phi(z) - \prod_{j=1}^{d}(z - \lambda_j)\| < \epsilon.$$

For $\phi \in \mathcal{P}_d(1)$, the worst case arithmetic complexity of this algorithm is

$$\mathcal{O}\big(d(\log d)^2 |\log \epsilon| + d^2(\log d)^2\big).$$

COROLLARY 2.2. *Let ϕ and ϵ be as in the theorem, and denote the roots of ϕ by ξ_j, represented with multiplicity. Then the algorithm can be used to produce the approximations λ_j satisfying*

$$|\xi_j - \lambda_j| < \epsilon,$$

with a worst-case arithmetic complexity of

$$\mathcal{O}\big(d^2(\log d)^2 |\log \epsilon| + d^2(\log d)^2\big).$$

Proof. This is an immediate consequence of the fact that if f and g are in $\mathcal{P}_d(1)$ with $\|f - g\| < (\epsilon/8d)^d$, then the roots of f and g are at most ϵ apart (See [Kim89b]). □

We prove the theorem as a series of lemmas in the following sections. There is a rough correspondence between the sections and the routines in the algorithm. Finally, we summarize all of these lemmas and give the proof in §2.8.

2.2. Selection of initial points. For each intermediate polynomial f_i, we need to select four sets of ($\deg f_i$) points at which to begin our iteration. These must be chosen so that the elements of each set map very near the same point in the target space, and the images of elements in successive groups are approximately $\frac{1}{4}$-turn apart. This can be accomplished either by evaluating f_i at a large number of points spaced around a circle in the source space, and then selecting from those, or by taking a much smaller number (perhaps as few as $4d$) of points and adjusting them with either a standard or modified Newton's method. The arithmetic complexity of either comes out much the same; we opt for the former method because of its conceptual simplicity.

The following lemma gives us bounds on how much the argument in the target space can vary between points around a circle containing all the roots in the source space. Using this, we see how many points are required to obtain our "good" points.

LEMMA 2.3. *Suppose all of the roots ζ of a polynomial f of degree d lie in $\mathbb{D}_R(0)$, and let*

$$\omega_m = 2R\,\mathrm{e}^{\frac{2\pi i m}{nd}}.$$

Then

$$\left|\arg \frac{f(\omega_{m+1})}{f(\omega_m)}\right| \le \frac{4\pi}{n}.$$

Proof. This argument appears in [Ren87, Lem. 7.1], although in a somewhat different form. We present an adapted version here. The idea is quite simple. Since

$$\arg \frac{f(\omega_{m+1})}{f(\omega_m)} = \arg \frac{\prod_{i=1}^{d}(\omega_{m+1} - \zeta_i)}{\prod_{i=1}^{d}(\omega_m - \zeta_i)} = \sum_{i=1}^{d} \arg \frac{\omega_{m+1} - \zeta_i}{\omega_m - \zeta_i},$$

we merely bound the angles in the source space and add them up.

Note that each root $|\zeta_i| < R$ and $|\omega_m| = 2R$, so

$$\begin{aligned}\left|\arg \frac{\omega_{m+1} - \zeta_i}{\omega_m - \zeta_i}\right| &\le 2\arctan \frac{|\omega_{m+1} - \omega_m|/2}{2R\cos\frac{\pi}{nd} - R} \\ &\le 2\arctan \frac{2R\sin\frac{\pi}{nd}}{2R(\cos\frac{\pi}{nd} - \frac{1}{2})} \\ &\le \frac{4\pi}{nd}.\end{aligned}$$

Refer to Fig. 2.1. Summing the d terms gives the result. □

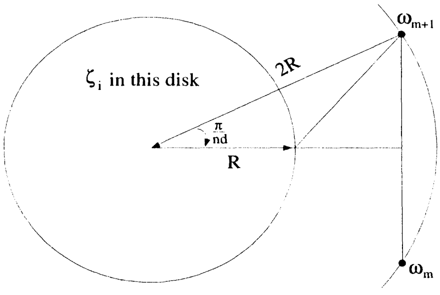


FIG. 2.1. *Calculation of the upper bound on the variation of argument in the target space. This picture is in the source space.*

From Theorem 1.5, for $h = \frac{1}{27}$ we require that our initial points be within $\sin^{-1}(\frac{h}{2}) < \frac{\pi}{169}$ of the central ray, so we start with $676d$ points equally spaced around the circle of radius $\frac{3}{2}$. We then evaluate the polynomial at each of them, and select four sets of d points whose arguments are closest to 0, $\frac{\pi}{2}$, π, and $\frac{3\pi}{2}$, respectively. For each of these, we take the initial target point w_0 to be the projection of its image onto the real or imaginary axis.

Remark. Some amount of computation can be saved if we use the same w_0 for all d elements of a given set of initial points, rather than just points with the same argument. This would make the computation of the w_n a scalar rather than a vector operation, that is, w_n would only need to be computed once for each group of d points. However, in order to do this, we must ensure that the images of each z_j in a set have approximately the same modulus as well as argument. This is best accomplished using some sort of Newton's method.

If an initial Newton's method is used, one can also choose a much smaller number of trial points ω_i. Such a method should converge well, since all the critical points of $f(z) - w_0$ are inside $\mathbb{D}_R$, while the roots of $f(z) - w_0$ and the ω_i are well outside.

2.3. Iteration of the path lifting method. In this section, we analyze the behavior of applying the path lifting method to a single well-chosen initial value z_0.

We first prove the theorem as promised in §1.3, and then we show that after a specified number of iterations, the result will be an approximate zero. Before doing this, we shall state the relevant special case of of [Kim88, Thm. 3.2], which gives a lower bound on how far a point moves under Newton's method.

LEMMA 2.4. *Let $\hat{z} = z - g(z)/g'(z)$, with $r = |g(z)|/R_g(z) < 0.148$, where $R_g(z)$ is the radius of convergence of g_z^{-1} as a power series based at z. Then*

$$\left|\frac{g(\hat{z})}{g(z)}\right| \leq B(r) \qquad where \qquad B(r) = 2r\frac{(1+r)^3}{(1-r)^5}.$$

Given this lemma, the proof of Theorem 1.5 is quite simple. Note that each step of the iteration in Theorem 1.5 corresponds to Newton's method applied to $f(z) - w_{n+1}$ at z_{n-1}.

THEOREM 1.5. *Suppose that the branch of the inverse $f_{z_0}^{-1}$ is analytic on a wedge $\mathcal{W}_{A,w_0}$, with $0 \leq A \leq \frac{\pi}{2}$, and let $h \leq \frac{\sin A}{19}$. Suppose also that $|f(z_0) - w_0| < h|w_0|/2$, and define*

$$w_n = (1-h)^n w_0, \qquad z_{n+1} = z_n - \frac{f(z_n) - w_{n+1}}{f'(z_n)}.$$

Then $|f(z_n) - w_n| \leq h|w_n|/2$ and $z_{n+1} \in f_{z_0}^{-1}\left(\mathcal{W}_{A,w_n}\right)$.

Proof. We shall prove this by induction. All that is required is to establish the conclusion, given that $|f(z_{n-1}) - w_{n-1}| < h|w_{n-1}|/2$.

Note that

$$|f(z_{n-1}) - w_n| \leq |f(z_{n-1}) - w_{n-1}| + |w_{n-1} - w_n| \leq 3h|w_{n-1}|/2,$$

and that

$$R_{f-w_n}(z_{n-1}) = R_f(z_{n-1}) \geq |w_n| \sin A - h|w_n|/2.$$

Since $h \leq \frac{\sin A}{19}$, we can apply Lemma 2.4 with $g(z) = f(z) - w_n$ and $r = \frac{3}{37}$ to obtain

$$\left|\frac{f(z_n) - w_n}{w_n}\right| = \left|\frac{f(z_n) - w_n}{f(z_{n-1}) - w_n}\right| \cdot \left|\frac{f(z_{n-1}) - w_n}{w_{n-1}(1-h)}\right| \leq B(r)\frac{3h}{2(1-h)}.$$

Because $B(\frac{3}{37}) < (1-h)/3$, we have our conclusion. □

Remark. The denominator of 19 in the upper bound on h can be relaxed only slightly in this proof. If we take $h \leq \frac{\sin A}{k(A)}$, then $k(A)$ is a monotonically increasing function, with $18.3096 \leq k(A) \leq 18.895$. Also, if $A > \frac{\pi}{2}$, we can take $h = \frac{1}{19}$.

We apply the path lifting method until we have an approximate zero, at which time we can use the standard Newton iteration. Since we are interested in finding τ-roots of f, we stop the iteration once we have an approximate zero for $f - \tau\frac{w_0}{|w_0|}$. Note that, as discussed in §2.1, this translation is necessary, since there are no approximate zeros in a neighborhood of a multiple root. Having approximate zeros for the perturbed polynomial gives us the following bound on the number of iterations required, and enables the weeding out of duplicates.

LEMMA 2.5. *Let $h \leq \frac{\sin A}{19}$, with w_i and z_i as in Theorem 1.5. If*

$$N = \left\lfloor \frac{\log_2(\tau/|w_0|)}{\log_2(1-h)} \right\rfloor,$$

then z_N is an approximate zero for $f - \tau\frac{w_0}{|w_0|}$. Furthermore, if

$$\hat{z} = z_N - \frac{f(z_N) - \tau\frac{w_0}{|w_0|}}{f'(z_N)}$$

then we have $\alpha(\hat{z}) \leq \frac{1}{8}$.

Proof. For notational convenience, set $\vec{\tau} = \tau\frac{w_0}{|w_0|}$.

By assumption, $f_{z_0}^{-1}$ is analytic on the wedge $\mathcal{W}_{A,w_0}$, and the initial point z_0 is close enough to w_0 that we can apply Theorem 1.5. Note that if N is as specified, we have

$$\tau \leq |w_N| \leq \frac{\tau}{1-h},$$

since $|w_N| = (1-h)^N|w_0|$ by definition. Also, $|f(z_N) - w_N| \leq h|w_N|/2$. Refer to Fig. 2.2.

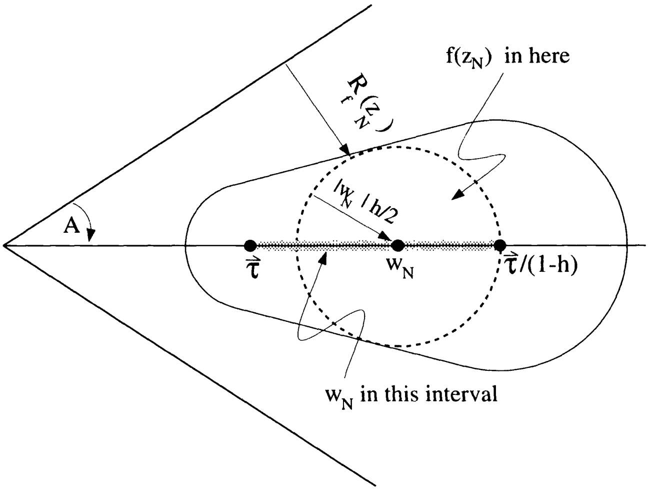


FIG. 2.2. *The locations of w_N and $f(z_N)$ in the target space when z_0 is in a "good quadrant."*

In order to conclude that z_N is an approximate zero, we need to show that we have $|f(z_N) - \vec{\tau}|/R_f(z_N) < \frac{1}{10}$, where $R_f(z_N)$ is the radius of convergence of $f_{z_N}^{-1}$. Consider $f(z_N)$ in some circle of radius $h|w|/2$, where $\arg w = \arg w_0$, and $|w| > \tau$. Then we have

$$\frac{|f(z_N) - \vec{\tau}|}{R_f(z_N)} \leq \frac{|w| + h|w|/2 - \tau}{|w|\sin A - h|w|/2} = \frac{2+h}{2\sin A - h} - \frac{\tau}{|w|(\sin A - h/2)}.$$

The maximum of this quantity occurs when $|w|$ is as large as possible, which in our case is $|w| = \tau/(1-h)$. Hence

$$\frac{|f(z_N) - \vec{\tau}|}{R_f(z_N)} \leq \frac{3h}{2\sin A - h} \leq 3/37 < 1/10.$$

By Lemma 1.4, z_N is an approximate zero of $f(z) - \vec{\tau}$.

In order to ensure that $\alpha < \frac{1}{8}$, we need to apply Newton's method once. Set

$$\hat{z} = z_N - \frac{f(z_N) - \vec{\tau}}{f'(z_N)}.$$

Then this point satisfies

$$\frac{\left|f(\hat{z}) - \vec{\tau}\right|}{|f(z_N) - \vec{\tau}|} \leq B(3/37) \leq 0.31271,$$

and so

$$\left|f(\hat{z}) - \vec{\tau}\right| \leq 0.31271\,(|f(z_N) - w_N| + |w_N| - \tau) \leq 0.31271\frac{3h\tau}{2(1-h)} \leq \frac{\tau\sin A}{39}.$$

This gives us

$$\frac{|f(\hat{z}) - \vec{\tau}|}{R_f(\hat{z})} \le \frac{\tau \sin A/39}{\tau \sin A - \tau \sin A/39} = 1/38.$$

As a consequence of Lemma 1.4, we have $\alpha(\hat{z}) < \frac{1}{8}$. □

2.4. Refinement of the root approximations. The routine `iterate-PLM` outputs a set of points $\mathcal{Y} = \{y_1, y_2, \ldots, y_d\}$ that may be approximate zeros for $\psi(z) = f(z) - \vec{\tau}$. We can use α_ψ (see Lemma 1.3) to choose those y_i that are indeed approximate zeros; we discard those y_i for which $\alpha_\psi(z) > \frac{1}{8}$. As a consequence of Lemmas 1.4 and 2.5, those y_i that started in the "good sector" will not be discarded.

In addition, we also want to ensure that we approximate each root only once (counted with multiplicity). Lemma 2.7 gives us conditions that allow us to weed out any duplicates; the proof relies on a variant of the Koebe Distortion Theorem which we quote here from [Kim88, Lem. 3.3].

LEMMA 2.6. *Let g be univalent on $\mathbb{D}_R(z)$. Then for any $s < 1$, we have*

$$\mathbb{D}_{tR}(g(z)) \subset g(\mathbb{D}_{sR}(z)) \subset \mathbb{D}_{uR}(g(z)),$$

where $t = s|g'(z)|/(1+s)^2$ and $u = s|g'(z)|/(1-s)^2$.

LEMMA 2.7. *Suppose that y_1 and y_2 are approximate zeros of $\psi(z)$, with $|\psi(y_1)| \ge |\psi(y_2)|$, and $R_\psi(y_j) \ge 12\psi(y_j)$, where $R_\psi(y_j)$ is the radius of convergence of $\psi_{y_j}^{-1}$. Then y_1 and y_2 approximate the same simple root ξ of ψ if and only if*

$$|y_1 - y_2| < \left|\frac{3\psi(y_1)}{\psi'(y_1)}\right|.$$

Proof. If y_1 and y_2 approximate the same simple root, then we have $\psi_{y_1}^{-1} = \psi_{y_2}^{-1}$. Since $|\psi(y_1) - \psi(y_2)| \le 2|\psi(y_1)|$, we have $\psi(y_2) \in \mathbb{D}_{2|\psi(y_1)|}(\psi(y_1))$. Thus we can apply Lemma 2.6 with $g = \psi^{-1}$, which is univalent on the disk of radius $R = 12|\psi(y_1)|$. Taking $s = \frac{1}{6}$, we have

$$\psi^{-1}\left(\mathbb{D}_{2|\psi(y_1)|}(\psi(y_1))\right) \text{ contained in the disk of radius } \frac{72|\psi(y_1)|}{25|\psi'(y_1)|} \text{ about } y_1,$$

so

$$|y_1 - y_2| < \frac{3|\psi(y_1)|}{|\psi'(y_1)|}.$$

For the other direction, we apply the Koebe $\frac{1}{4}$-Theorem (or Lemma 2.6 with $s = 1$; it is the same thing) to see that $\psi_{y_1}^{-1}\left(\mathbb{D}_{12|\psi(y_1)|}(\psi(y_1))\right)$ contains the disk of radius $3|\psi(y_1)|/|\psi'(y_1)|$ about y_1. Thus, if the distance between y_1 and y_2 is less than this amount, we must have $\psi_{y_1}^{-1}(\psi(y_2)) = y_2$, that is, y_1 and y_2 approximate the same root of ψ. □

Remark. Note that $\alpha_\psi(z) < \frac{1}{8}$ is not sufficient to imply that the hypotheses of Lemma 2.7 are satisfied, since this only gives $R_\psi(z) \ge 4|\psi(z)|/3$. However, if Newton's method is applied to such a point at least 3 times, the value of $|\psi|$ will decrease by at least $\frac{1}{128}$, and so the lemma can be applied. Since we need to "polish" the approximations with Newton's method in order to control the error in the deflation, we do that before weeding out the duplicates. The total number of iterations of Newton's method required is calculated in §2.6, but it is greater than three in all cases. In practice, one should probably perform the weeding in the routine `select-approx-zeros`, in order to avoid polishing points that will be discarded later.

2.5. Deflation of intermediate polynomials. Here we compute an explicit bound on the error introduced by the deflation step. We start with a polynomial ψ which has roots $\{\xi_j\}_{j=1,\dots,d}$, and a set of approximations to these roots that we denote by $\{v_k\}_{k=1,\dots,n}$ with $n \leq d$. We then use polynomial interpolation via the discrete Fourier matrix to obtain a polynomial q of degree $d-n$ so that

$$\psi(z) \approx p(z)q(z), \qquad \text{where} \qquad p(z) = \prod_{k=1}^{n}(z - v_k).$$

In this section, we estimate $\|\psi(z) - p(z)q(z)\|$, as well as the accumulated error in repeating this process until $q(z)$ is a constant.

Recall that the finite Fourier matrix $\mathcal{M}$ has as its j, kth entry the jkth power of a primitive $d+1$st-root of unity. That is,

$$m_{j,k} = \omega^{jk} = \mathrm{e}^{2\pi \mathrm{i} jk/(d+1)} \qquad j,k = 0,\dots,d.$$

Then one can readily see that if $f(z) = \sum_{j=0}^{d} a_j z^j$ and $\mathcal{A}$ is the column vector of the coefficients of f, then the product $\mathcal{MA}$ will be the vector $\mathcal{B}$ whose jth entry is the value of $f(\omega^j)$. Also, given the values of f evaluated at the powers of ω, we can easily compute the coefficients of f as the product $\mathcal{M}^{-1}\mathcal{B}$.

Our first lemma gives us estimates on the size of the error caused by a single deflation. As is common with this sort of thing, the proof is neither terribly entertaining or enlightening.

LEMMA 2.8. *Suppose $\psi(z) = \prod_{j=1}^{d}(z-\xi_j)$, with $|\xi_j| < \frac{3}{4}$, factors as $\psi(z) = P(z)Q(z)$, where $\deg(P) = n < d$ and $\deg(Q) = m = d-n$. Let $p(z) = \prod_{j=1}^{n}(z - v_j)$, with $|\xi_j - v_j| < \delta < 1/8d^2$, and define $q(z)$ and $r(z)$ by $\psi(z) = p(z)q(z) + r(z)$, where q is found by polynomial interpolation as described above. Then*

$$\begin{aligned} \|P(z) - p(z)\| &< 8n\delta\,(7/4)^n \\ \|Q(z) - q(z)\| &< 8m\delta\,(7/4)^m \\ \|r(z)\| &< 8d\delta\,(7/4)^d\,. \end{aligned}$$

Proof. Let $\omega = \mathrm{e}^{2\pi\mathrm{i}/(n+1)}$, and let $\mathcal{B}$ be the vector with entries $b_j = P(\omega^j) - p(\omega^j)$, for $0 \leq j \leq n$. Note that

$$\begin{aligned} b_j &= \prod_{k=1}^{n}(\omega^j - \xi_k) - \prod_{k=1}^{n}(\omega^j - v_k) \\ &= \left(\prod_{k=1}^{n}(\omega^j - \xi_k)\right)\left(1 - \prod_{k=1}^{n}\frac{\omega^j - v_k}{\omega^j - \xi_k}\right) \\ &= \left(\prod_{k=1}^{n}(\omega^j - \xi_k)\right)\left(1 - \prod_{k=1}^{n}\left(1 + \frac{\xi_k - v_k}{\omega^j - \xi_k}\right)\right). \end{aligned}$$

Also note that

$$\frac{1}{4} < |\omega^j - \xi_k| < \frac{7}{4} \qquad \text{and} \qquad \left|\frac{\xi_k - v_k}{\omega^j - \xi_k}\right| \leq 4\delta.$$

Since $4\delta \leq 1/2d^2 < 1/2n^2$, we have $(1+4\delta)^n \leq 8n\delta + 1$, and so

$$|b_j| < (7/4)^n\big((1+4\delta)^n - 1\big) < 8n\delta\,(7/4)^n.$$

Thus

$$\|P(z) - p(z)\| = \|\mathcal{M}^{-1}\mathcal{B}\| \leq \|\mathcal{B}\| < 8n\delta\,(7/4)^n.$$

We have a similar computation for the bound on $\|Q - q\|$: Let $\eta = e^{2\pi i/(m+1)}$ and let $\mathcal{C}$ be the vector with entries $c_j = Q(\eta^j) - q(\eta^j)$, $j = 0, \dots, m$. Then

$$c_j = \frac{\psi(\eta^j)}{P(\eta^j)} - \frac{\psi(\eta^j)}{p(\eta^j)} = \frac{\psi(\eta^j)}{P(\eta^j)}\left(1 - \frac{P(\eta^j)}{p(\eta^j)}\right) = Q(\eta^j)\left(1 - \prod_{k=1}^{m}\frac{\eta^j - v_k}{\eta^j - \xi_k}\right).$$

As in the case of $P(z) - p(z)$, we obtain

$$\|Q(z) - q(z)\| = \|\mathcal{M}^{-1}\mathcal{C}\| \leq \|\mathcal{C}\| < 8m\delta\,(7/4)^m.$$

Finally, for the bound on $\|r(z)\|$, note that $r(z) = P(z)Q(z) - p(z)q(z)$. If we write $r(z) = \sum_{j=0}^{d} r_j z^j$, $P(z) = \sum_{j=0}^{d} P_j z^j$, and so on, then we have

$$r_j = \sum_{k=0}^{j} P_{j-k}Q_k - \sum_{k=0}^{j} p_{j-k}q_k = \sum_{k=0}^{j}\big(q_k(P_{j-k} - p_{j-k}) - P_{j-k}(Q_k - q_k)\big).$$

Since $|\xi_j| < \frac{3}{4}$, we have the following crude bounds on the coefficients of P and q:

$$|P_{j-k}| \leq (7/4)^n \qquad \text{and} \qquad |q_j| \leq (7/4)^m.$$

Combining this with the bounds on $\|P - p\|$ and $\|Q - q\|$, we get

$$\begin{aligned} |r_j| &\leq \sum_{k=0}^{j}\big(8m\delta\,(7/4)^m\,(7/4)^n + 8n\delta\,(7/4)^n\,(7/4)^m\big) \\ &\leq 8jd\delta\,(7/4)^d \\ &\leq 8d^2\delta\,(7/4)^d. \qquad \square \end{aligned}$$

Now that we have bounds on the error in one step of deflation, we can bound the error introduced by repeated deflation. We assume that our initial polynomial f has roots in $\mathbb{D}_{1/2}$ so that we can ensure that the roots of the subsequent polynomials f_k remain in $\mathbb{D}_{3/4}$ as required by Lemma 2.8.

LEMMA 2.9. *Suppose* $f(z) = \prod_{j=1}^{d}(z - \zeta_j)$, *with* $|\zeta_j| < \frac{1}{2}$. *Let* $f_0 = f$, *and define*

$$f_k(z) = p_{k+1}(z)f_{k+1}(z) + r_{k+1}(z)$$

for $0 < k < m - 1$, *where* p_j, f_j, *and* r_j *are determined by polynomial interpolation as in Lemma* 2.8, *with* $f_m(z) = 1$. *Suppose also that* $\deg r_{k+1} < \deg f_k = d_k$, *and that*

$$\|r_k\| < \min\left(\frac{1}{4(4d)^d}, \mu(4/7)^{d+3}\right).$$

Then

$$\|f - p_1p_2\cdots p_m\| < \mu.$$

Furthermore, if we have $d_k \leq d_{k-1}/2$ *and* $\mu \leq (7/16)^d$, *we need only require that* $\|r_k\| < \mu(4/7)^{d+3}$.

Proof. Note that $f - p_1p_2\cdots p_m = p_1\cdots p_m r_{m-1} + \cdots + p_1r_2 + r_1$, and so

$$\|f - p_1p_2\cdots p_m\| \leq \|p_1\cdots p_{m-1}\|\|r_{m-1}\| + \cdots + \|p_1\|\|r_2\| + \|r_1\|.$$

First, we show that $\|p_1\cdots p_k\| \leq (7/4)^{\deg p_1\cdots p_k}$. We shall use induction to show that the roots of $p_1\cdots p_k$ are always in the circle of radius $\frac{1}{2}+\frac{k}{4d} < \frac{3}{4}$. On the circle $|z| = \frac{1}{2}+\frac{k}{4d}$, we have

$$|f_{k-1}(z) - p_k(z)f_k(z)| \leq \|r_k\| \sum_{j=0}^{\deg r_k} |z|^j \leq \frac{d}{(4d)^d(2d-k)} \leq \frac{1}{(4d)^d} \leq |f_{k-1}(z)|,$$

and so by Rouché's Theorem the roots of p_kf_k lie inside $\mathbb{D}_{\frac{1}{2}+\frac{k}{4d}}$. Thus the roots of $p_1p_2\cdots p_k$ also lie in this disk, and so the coefficients are less than $(7/4)^{\deg p_1\cdots p_k}$.

Since $\deg p_1p_2\cdots p_k < d-(m-k)$, we can conclude that

$$\begin{aligned}\sum_{k=1}^{m-1} \|p_1p_2\cdots p_k\|\|r_{k+1}\| &\leq \sum_{k=1}^{m-1} (7/4)^{d-(m-k)}\,(4/7)^{d+3}\,\mu \\ &\leq \frac{4}{3}\,(7/4)^{d+2}\,(4/7)^{d+3}\,\mu \\ &< \mu.\end{aligned}$$

If we require that $d_k \leq d/2^k$, that is, we find at least half of the roots at each step, then we can relax the restriction on $\|r_k\|$ somewhat. If we have $\|r_k\| \leq 1/(4\cdot 4^d)$, then we can use Rouché's Theorem as before to show that the roots of p_kf_k are in the disk of radius $C_k = \frac{1}{2}+\frac{1}{2}\sum_{n=1}^k 2^{-2^k} < 3/4$. Note that for $|z| = C_k$, we have

$$|f_{k-1}(z)| \geq (C_k - C_{k-1})^{d_{k-1}} \geq \left(\frac{1}{2\cdot 2^{2^k}}\right)^{d/2^{k-1}} \geq 1/4^d.$$

Applying almost the same argument as before, we have

$$|f_{k-1}(z) - p_k(z)f_k(z)| \leq \|r_k\| \sum_{j=0}^{\deg r_k} |z|^j \leq \frac{1}{4^d} \leq |f_{k-1}(z)|.$$

Since $\|r_k\| < \mu(4/7)^{d+3} < 1/(4\cdot 4^d)$, we are done. □

2.6. Controlling the error. In this section, we compute the size of τ that we can use to ensure we have an ϵ-factorization of ψ. The following very simple proposition shows that if the norms of two polynomials are close, so are the norms of the rescaled versions. This gives us the relationship between ϵ and the number μ used in Lemma 2.9.

PROPOSITION 2.10. *Suppose that ϕ is a monic polynomial with all its roots in $\mathbb{D}_R$, so that the roots of $f(z) = (2R)^{-d}\phi(2Rz)$ are in $\mathbb{D}_{1/2}$. Then if $\|f - p\| \le \epsilon$, we have*

$$\left\|\phi(z) - (2R)^d p\left(\frac{z}{2R}\right)\right\| \le (2R)^d \epsilon.$$

Proof. Let $f(z) = \sum a_j z^j$ and $p(z) = \sum b_j z^j$. Then $\phi(z) = \sum (2R)^{d-j} a_j z^j$ and $(2R)^d p\left(\frac{z}{2R}\right) = \sum (2R)^{d-j} b_j z^j$. Since $\max|\sum (2R)^{d-j}(a_j - b_j)| \le (2R)^d |a_j - b_j|$, we have our claim. □

Our input polynomial ϕ is in $\mathcal{P}_d(1)$ and f is the rescaled polynomial as in the previous proposition, so $R = 2$. Then an ϵ-factorization of ϕ corresponds to an $\epsilon/4^d$-factorization of f, so we take $\mu = \epsilon/4^d$.

In order to properly approximate the roots of f, we need to ensure that the remainder r_k (as in Lemma 2.9) at the kth step satisfies $\|r_k\| < 2\tau$, where $\tau = (4/7)^{d+3}\mu/2 = 32\epsilon/7^{d+3}$.

At each stage, we translate f_k by τ, and ensure that the error introduced by the deflation of the translated polynomial is no more than τ. By Lemma 2.8, we need the root distance between translated polynomial and the deflated polynomial to satisfy

$$\delta \le \frac{\tau}{8d}\left(\frac{4}{7}\right)^d.$$

Then we will have

$$\|r_k\| = \|f_{k-1} - p_k f_k\| \le \|(f_{k-1} - \vec{\tau}) - p_k f_k\| + \|(f_{k-1} - \vec{\tau}) - f_{k-1}\| \le \tau + \tau.$$

In order to achieve the root distance less than δ, we apply Newton's method to the approximate zeros found by the routine `iterate-PLM` (see Lemma 2.5). Since each point z is an approximate zero to the root ξ of the translated polynomial, we have by Definition 1.2

$$\delta \le 8\left(\frac{1}{2}\right)^{2^n} |z - \xi|.$$

Thus iterating Newton's method $\log_2 \log_2(\frac{8}{\delta})$ times, as is done in `polish-roots`, will give the desired result.

2.7. Arithmetic complexity. In this section we count the number of arithmetic operations involved in using the algorithm to obtain an ϵ-factorization of a polynomial in $\mathcal{P}_d(1)$.

In the main routine, we rescale the polynomial and then call `get-half-roots-and-deflate` at most $\log_2 d$ times, since at least half of the roots are found in each call.

- The cost of rescaling is $2d$ multiplications.
- Each call to `get-half-roots-and-deflate` calls the routine `choose-4d-good-initial-points` once, and makes at most four calls to each of `iterate-PLM`, `select-approx-zeros`, `polish-roots`, and `weed-out-duplicates`, and one call to `deflate`. As before, we denote the degree of the input polynomial by d and the degree of the kth intermediate polynomial by d_k.
  - The routine `choose-4d-good-initial-points` involves evaluation of f_k at $676d_k$ points, and $676d_k$ comparisons. Since we can evaluate f_k at m points with a cost of $\mathcal{O}\big(m(\log d_k)^2\big)$ operations (see [BM75]), this gives a total of $\mathcal{O}\big(d_k(\log d_k)^2\big)$ operations.
  - For `iterate-PLM`, each iteration evaluates

$$z_n - \frac{f(z_n) - w_{n+1}}{f'(z_n)}$$

at d_k points, which costs $\mathcal{O}\big(d_k(\log d_k)^2\big)$. This is done N times, where $N < 27\log\left(\frac{|w_0|}{\tau}\right)$. Note that since the roots of f are in $\mathbb{D}_{3/4}$ and $|z_0| = \frac{3}{2}$, we have $|w_0| = |f_k(z_0)| \leq (9/4)^{d_k}$. Since $\tau = 32\epsilon/7^{d+3}$, we have $N = C_1(d + |\log\epsilon|)$ for some constant C_1. This gives a total of

$$\mathcal{O}\big((d+|\log\epsilon|)(d_k(\log d_k)^2)\big)$$

operations.

- For `select-approx-zeros`, we need to evaluate α at d_k points, which requires evaluation of all the derivatives of f. This requires $\mathcal{O}\big(d_k^2(\log d_k)^2\big)$ operations.
- The routine `polish-roots` performs M iterations of Newton's method at d_k points. Since $M = C_2(\log d + \log|\log\epsilon|)$ for some constant C_2, we have a total operation count of

$$\mathcal{O}\big((d_k(\log d_k)^2)(\log d + \log|\log\epsilon|)\big).$$

- `Weed-out-duplicates` requires a sort of at most d_k points, which costs $\mathcal{O}(d_k\log d_k)$, and evaluation of f and f' at d_k points. Thus the total cost of this routine is $\mathcal{O}\big(d_k(\log d_k)^2\big)$.
- Lastly, `deflate` costs $\mathcal{O}\big(d_k(\log d_k)^2\big)$ operations.

Thus the overall cost of each call to `get-half-roots-and-deflate` is dominated by that of `iterate-PLM`, and is at most $\mathcal{O}\big(d_k(\log d_k)^2(d+|\log\epsilon|)\big)$ operations. Since $d_{k+1} \leq d_k/2$, we have a total cost of at most

$$\mathcal{O}\big(d^2(\log d)^2 + d(\log d)^2|\log\epsilon|\big)$$

operations to obtain an ϵ-factorization of the polynomial.

2.8. Summary and proof of main theorem. At this point, we have actually already proven Theorem 2.1, but we would like to tie together the various steps involved. Just to refresh your memory, this theorem says, in essence, that our algorithm always produces an ϵ-factorization with the stated complexity.

Recall that the algorithm performs the approximate factorization in stages using the routine `get-half-roots-and-deflate`; at the kth step, we produce a function f_k and sets of approximations Λ_j so that

$$f(z) \approx f_k(z)\prod_{\lambda_i\in\Lambda}(z-\lambda_i),$$

where $\Lambda = \bigcup_{j\leq k}\Lambda_j$. In order to prove Theorem 2.1, we need to show that the number of approximations found at the kth stage $(\#\Lambda_k)$ is at least $(\deg f_{k-1})/2$, and that $\|f(z) - f_k(z)\prod_{\lambda_i\in\Lambda}(z-\lambda_i)\| \leq \epsilon/4^d$. Given this, the complexity calculation in the previous section will apply, and we shall have the result.

First, note that as a consequence of Lemma 1.6, there will always be a quarter-plane in the target space on which at least $d_k/2$ branches of f_k^{-1} are defined. This means that if we start with d_k points z_j that are well-spaced in the source space (so that each sheet in the target space is represented), then for at least half of them there will be a branch of the inverse $f_{z_j}^{-1}$ defined on the entire quadrant, and $f_{z_i}^{-1} \neq f_{z_j}^{-1}$. For these points z_j, if we ensure that $f(z_j)$ is close enough to the center line of the quadrant, at least half of them will satisfy the hypothesis of Theorem 1.5 and so the routine `iterate-PLM` will produce approximations to each of the corresponding $d_k/2$ roots of f_k (with multiplicity). Such initial points z_j will be produced by `choose-4d-good-initial-points`, as was shown in §2.2. As a consequence of Lemma 2.5, the good approximations `iterate-PLM` are approximate zeros of $\psi = f_k - \vec{\tau}$, with $\alpha_\psi < \frac{1}{8}$. As was discussed in §2.4, application of the routines `select-approx-zeros` and `weed-out-duplicates` will select exactly one representative for each approximated root of ψ, giving at least $d_k/2$ such approximations λ_j. This selection is necessary since some of the initial z_j that did not satisfy the hypothesis of Theorem 1.5 may still have converged.

As was shown in §2.6, the approximations produced yield an $\epsilon/4^d$-factorization, since the each the λ_i are made sufficiently close to the roots of ψ by the routine `polish-roots`, and $\|\psi - f_k\|$ is sufficiently small. This completes the proof of Theorem 2.1.

REFERENCES

[Ahl79] L. Ahlfors, *Complex Analysis*, McGraw-Hill, New York, 1979.

[BFKT88] M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, *A fast parallel algorithm for determining all roots of a polynomial with real roots*, SIAM J. Comput., 17 (1988), pp. 1081–1092.

[BT90] M. Ben-Or and P. Tiwari, *Simple algorithms for approximating all roots of a polynomial with real roots*, J. Complexity, 6 (1990), pp. 417–442.

[BM75] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York, 1975.

[DH69] B. Dejon and P. Henrici, eds., *Constructive Aspects of the Fundamental Theorem of Algebra*, Wiley-Interscience, New York, 1969.

[Fri90] J. Friedman, *Random polynomials and approximate zeros of Newton's method*, SIAM J. Comput., 19 (1990), pp. 1068–1099.

[Hen74] P. Henrici, *Applied and Computational Complex Analysis*, Wiley-Interscience, New York, 1974.

[JT70] M. A. Jenkins and J. F. Traub, *A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration*, Numer. Math., 14 (1970), pp. 252–263.

[Kim88] M. Kim, *On approximate zeros and rootfinding algorithms for a complex polynomial*, Math. Comp., 51 (1988), pp. 707–719.

[Kim89a] ———, *Error analysis and bit complexity: polynomial root finding problems* I, Bellcore, Morristown, NJ, 1989, preprint.

[Kim89b] ———, *Topological complexity of a root finding algorithm*, J. Complexity, 5 (1989), pp. 331–344.

[Man92] A. Manning, *How to be sure of solving a complex polynomial using Newton's method*, Bol. Soc. Brasil Mat., 22 (1992), pp. 157–177.

[Mar66] M. Marden, *The Geometry of Polynomials*, American Mathematical Society, Providence, RI, 1966.

[Nef90] A. Neff, *Specified precision polynomial root isolation in NC*, in 31st FOCS, IEEE, Washington, DC, 1990, pp. 152–162.

[Pan87] V. Pan, *Sequential and parallel complexity of approximate evaluation of polynomial zeros*, Comput. Math. Appl., 14 (1987), pp. 561–622.

[Ren87] J. Renegar, *On the worst-case arithmetic complexity of approximating zeros of polynomials*, J. Complexity, 3 (1987), pp. 90–113.

[Sch82] A. Schönhage, *The fundamental theorem of algebra in terms of computational complexity*. Tech. report, Mathematisches Institut, Universität Tübingen, Germany, 1982.

[Shu83] M. Shub, *The geometry and topology of dynamical systems and algorithms for numerical problems*, Notes from lectures given at D.D.4 Peking University, Beijing, China, 1983.

[SS85] M. Shub and S. Smale, *Computational complexity: on the geometry of polynomials and a theory of cost* I, Ann. Sci. École Norm. Sup., 18 (1985), pp. 107–142.

[SS86] ———, *Computational complexity: on the geometry of polynomials and a theory of cost* II, SIAM J. Comput., 15 (1986), pp. 145–161.

[SS93a] ———, *Complexity of Bezout's theorem* I: *geometric aspects*, J. Amer. Math. Soc, 6 (1993), pp. 459–501.

[SS93b] ———, *Complexity of Bezout's theorem* II: *volumes and probabilities*, in Computational Algebraic Geometry, F. Eyssette and A. Galligo, eds., Progress in Mathematics, Vol. 109, Birkhauser, 1993, pp. 267–285.

[SS93c] M. Shub and S. Smale, *Complexity of Bezout's theorem* III: *condition number and packing*, J. Complexity, 9 (1993), pp. 4–14.

[SS93d] ———, *Complexity of Bezout's theorem* IV: *probability of success, extensions*. IBM research report, 1993.

[STW88] M. Shub, D. Tischler, and R. Williams, *The Newtonian graph of a complex polynomial*, SIAM J. Math. Anal., 19 (1988), pp. 246–256.

[Sma81] S. Smale, *The fundamental theorem of algebra and complexity theory*, Bull. Amer. Math. Soc., 4 (1981), pp. 1–36.

[Sma85] ———, *On the efficiency of algorithms of analysis*, Bull. Amer. Math. Soc., 13 (1985), pp. 87–121.

[Sma86] ———, *Newton's method estimates from data at one point*, in The Merging Disciplines: New Directions in Pure, Applied, and Computational Mathematics, Springer-Verlag, Berlin, New York, 1986, pp. 185–196.

[Sut89] S. Sutherland, *Finding roots of complex polynomials with Newton's method*, Ph.D. thesis, Boston University, Boston, MA, 1989.

[Tis89] D. Tischler, *Critical points and values of complex polynomials*, J. Complexity, 5 (1989), pp. 438–456.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 437–446, April 1994

COMPUTING AND VERIFYING DEPTH ORDERS*

MARK DE BERG†, MARK OVERMARS†, AND OTFRIED SCHWARZKOPF†

Abstract. A depth order on a set of line segments in 3-space is an order such that line segment a comes before line segment a' in the order when a lies below a' or, in other words, when there is a vertical ray that first intersects a' and then intersects a. Efficient algorithms for the computation and verification of depth orders of sets of n line segments in 3-space are presented. The algorithms run in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If all line segments are axis-parallel or, more generally, have only a constant number of different orientations, then the sorting algorithm runs in $O(n \log^3 n)$ time and the verification takes $O(n \log^2 n)$ time. The algorithms can be generalized to handle triangles and other polygons instead of line segments. They are based on a general framework for computing and verifying linear orders extending implicitly defined binary relations.

Key words. computational geometry, depth orders, three dimensions, linear extensions of partial orders

AMS subject classification. 68Q25

1. Introduction.

Hidden surface removal is an important problem in computer graphics. In a typical setting, we are given a set of nonintersecting polyhedral objects in 3-space and a view point and want to compute which parts of the objects can be seen from the view point.

An efficient way of solving this problem is the *painter's algorithm*; see, for example, [10]. In this algorithm one tries to "paint" the objects in a back-to-front order onto the screen. Thus the objects in the front are painted on top of the objects in the back, resulting in a correct view of the scene. Such a back to front ordering is called a *depth order* of the set of objects. Note that a depth order does not always exist, since there can be *cyclic overlap* among the objects, as is the case for the three triangles shown in Fig. 1. A closely related approach uses a binary space partition tree to obtain a displaying order for the objects in a scene [11]. A binary space partition (BSP) cuts the objects in such a way that there is a depth order in any direction. Unfortunately, the number of fragments and, hence, the size of the resulting BSP tree can be as large as $\Omega(n^2)$ [20]. Hence, this approach can be very wasteful if there is no cyclic overlap in the viewing direction.

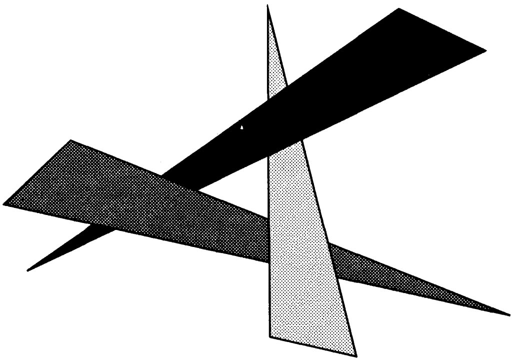


FIG. 1. *Cyclic overlap among triangles.*

The view of a scene consists of a subdivision of the viewing plane into maximal connected regions such that in each region either (some portion of) a single object can be seen or no

*Received by the editors December 16, 1991; accepted for publication (in revised form) April 20, 1993. This research was supported by the ESPRIT Basic Research Action 3075 (project Algorithms and Complexity). The first and second authors were also supported by the Dutch Organization for Scientific Research (N.W.O.).

†Department of Computer Science, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

object is seen. Sometimes it is necessary to compute a combinatorial representation of this so-called *visibility map*. Note that the painter's algorithm does not give us such a combinatorial representation. The combinatorial complexity of the visibility map of a set of objects with n edges in total varies between $O(1)$ and $\Omega(n^2)$. Hence, it would be nice to have an *output-sensitive* algorithm, that is, an algorithm whose running time is dependent on the complexity of the visibility map. Almost all output-sensitive algorithms known to date require that a depth order on the objects is given; see, for example, [14], [16], [19], [21]. Only the recent algorithms of [8], [9] do not need a depth order. The implementation of the latter algorithms, however, is much easier when a depth order is known.

It is thus important to be able to compute depth orders efficiently. This problem was studied by Chazelle et al. [5]. When the objects are lines in 3-space, they noted that a depth order can be obtained by a standard sorting algorithm, because any two lines can be compared (assuming no two have parallel projections). If there is cyclic overlap, however, then the outcome of the sorting algorithm is not a valid depth order. Verifying whether a depth order is valid is no trivial matter though; in [5], Chazelle et al. presented an $O(n^{4/3+\varepsilon})$ time algorithm to verify a given depth order of a set of lines. When the objects are line segments in 3-space, the problem becomes much harder, since not every pair of line segments can be compared. For this case, the best algorithm that was known runs in time $O(n \log n + k)$, where k is the number of intersections in the projection plane, or, in other words, the number of pairs that can be compared directly [5], [18]. Note that k can be $\Theta(n^2)$ and, hence, that the worst-case running time of these algorithms is $\Theta(n^2)$. Even for the case of axis-parallel line segments, it was an open problem to find a depth order in $o(n^2)$ time [21].

In this paper we show that a depth order for a set of line segments in 3-space can be computed in subquadratic time. More specifically, we give an algorithm that computes a depth order in time $O(n^{4/3+\varepsilon})$. We also present an algorithm that verifies a given order in $O(n^{4/3+\varepsilon})$ time. When the line segments are c-oriented, that is, they have only c different orientations for some constant c, then the sorting algorithm runs in $O(n \log^3 n)$ time and verification takes $O(n \log^2 n)$ time. Note that axis-parallel line segments are 3-oriented. The results can be generalized to depth orders for sets of triangles, or other polygons, instead of line segments.

The algorithms that we give are surprisingly simple. They are based on a general framework for computing a linear order extending a relation $(S, \prec)$. It is easy to compute an order in time that is linear in the number of pairs that are related; to this end one sorts the directed graph $\mathcal{G} = (S, E)$ topologically, where $(a, a') \in E$ if and only if $a \prec a'$. This is the approach taken in [5], [18] to sort a set S of n line segments: first compute all pairs of line segments that are related—this can be done in $O(n \log n + k)$ time by computing all intersections in the projection plane—and then sort the corresponding graph $\mathcal{G}$ in $O(n + k)$ time. Note that if $(S, \prec)$ does not contain a cycle then the sorting will succeed, otherwise some cycle will be detected in the graph $\mathcal{G}$. We show that it is not necessary to compute the full graph corresponding to $(S, \prec)$. All that is needed is to have a data structure that answers the following question: Given an element $a \in S$, return a predecessor of a and a successor of a, if they exist. The data structure should allow for the deletion of an element a in S in sublinear time. In cases where the relation is given implicitly—such as for depth orders—this is often possible. Our algorithm uses an interesting form of divide-and-conquer, where the divide-step does not need to be balanced. In fact, the more unbalanced it is, the better the running time of the algorithm.

There is some previous work on the computation of a linear order that extends a partial order. This work is also in the context of depth orders, in particular, depth orders in two-dimensional space [12], [13], [23] and depth orders for spheres in 3-space [22]. Unfortunately, the solutions given in these papers do not generalize to our setting. Of related interest is also a paper by Kenyon-Mathieu and King [15], who describe an algorithm that verifies whether a given partial order holds on n elements from an unknown total order.

The rest of this paper is organized as follows. In §2 we present our general framework for computing a linear order extending a relation $(S, \prec)$, and in §3 we give an algorithm to verify a given order. In §4 we show how to use these results to compute or verify a depth order for a set of line segments (or triangles, or polygons) in 3-space. We make some concluding remarks in §5.

2. Computing linear extensions. Let $\prec$ be a binary relation defined on a set S of n elements. Note that $\prec$ is not necessarily a partial order, since we do not assume transitivity. This will be useful in our application. In this section it is shown how to compute a linear order extending $(S, \prec)$ or to decide that $(S, \prec)$ contains a cycle. Thus we want to compute an order $a_1, \ldots, a_n$ on the elements in S such that $a_i \prec a_j$ implies $i < j$. The algorithm that we will give for this problem needs a data structure $\mathcal{D}_\prec$ for storing a subset $S' \subseteq S$ that can return a predecessor in S' of a query element $a \in S$. More formally, QUERY$(a, \mathcal{D}_\prec)$ returns an element $a' \in S'$ such that $a' \prec a$ or *NIL* if there is no such element. We call such a query a *predecessor query*. Similarly, we need a structure $\mathcal{D}_\succ$ for *successor queries*. To make our algorithm efficient, the structures should allow for efficient deletions of elements from S' and the preprocessing time should not be too high.

Let us define $\prec_*$ to be the transitive closure of $\prec$ and $\succ_*$ to be the transitive closure of $\succ$. The basic strategy of the algorithm is divide and conquer: we pick a pivot element $a_{\mathrm{piv}} \in S$, partition the remaining elements into a subset $S_\prec$ of elements a that must come before a_{piv} in the order because $a \prec_* a_{\mathrm{piv}}$ and a subset $S_\succ$ of elements that must come after a_{piv} in the desired order because $a_{\mathrm{piv}} \prec_* a$, and recursively sort these sets. Note that not every pair of elements is comparable under $\prec_*$. Hence, except for the subsets $S_\prec$ and $S_\succ$, there is a third subset $S_\approx$ of elements that cannot be compared to a_{piv} under $\prec_*$. This subset should be sorted recursively as well. To find the subsets $S_\prec$ and $S_\succ$ efficiently, the data structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$ are used. Consider the subset $S_\prec$. By querying $\mathcal{D}_\prec$ with element a_{piv}, we can find an element a such that $a \prec a_{\mathrm{piv}}$. We delete a from $\mathcal{D}_\prec$ to avoid reporting it more than once and query once more with a_{piv}. Continuing in this manner until the answer to the query is *NIL*, we can find all elements $a \in S$ such that $a \prec a_{\mathrm{piv}}$. However, we want to find all elements a such that $a \prec_* a_{\mathrm{piv}}$. Thus we also have to query $\mathcal{D}_\prec$ with the elements a that we have just found, query with the new elements that we find, and so forth. Whenever we find an element, it is deleted from $\mathcal{D}_\prec$ and we query with it until we have found all predecessors of it (that have not been found before). This way we can compute the set $S_\prec$ with a number of queries in $\mathcal{D}_\prec$ that is linear in the size of $S_\prec$. Notice that when we find a_{piv} as an answer to a query, there must be a cycle in the relation. The subset $S_\succ$ can be found in a similar way, using the data structure $\mathcal{D}_\succ$. The subset $S_\approx$ contains the remaining elements.

There is one major problem with this approach: we cannot ensure that the partitioning is balanced, that is, that the sets $S_\prec$, $S_\succ$, and $S_\approx$ have about the same size. Fortunately, we can circumvent this if we make the following two observations. First, we note that we need not treat the subset $S_\approx$ separately. We can put the elements of $S_\approx$ in either $S_\prec$ or $S_\succ$, as long as we do it consistently, that is, as long as we put all elements in the same set. It seems that this only makes things worse because the partitioning gets more unbalanced. But now we observe that it is enough to find the smaller of the two subsets $S_\prec$ and $S_\succ$. The remaining elements—which can be elements of $S_\approx$—are all put into one set. It is possible to find the smaller of the two subsets $S_\prec$ and $S_\succ$—without computing the complete larger set as well—with a number of queries that is linear in its size, by doing a "tandem search": alternatingly, find an element of $S_\prec$ and an element of $S_\succ$ until the computation of one of the two subsets has been completed. Thus we partition S into two subsets in time that is dependent on the size of the smaller of the two subsets. This means that the more unbalanced the partitioning is, the faster it is performed, leading to a good worst-case running time for the algorithm. There is one problem left that we have not addressed so far: we cannot afford to build the data structures that we need for the recursive call for the large set from scratch. Fortunately, we can obtain these structures from those that we have at the end of the tandem search by reinserting and deleting certain elements.

The algorithm for computing an ordering on $(S, \prec)$ first builds the data structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$ on the set S and then calls the procedure ORDER, with the set S and these two data structures as arguments. Below follows a detailed description of this procedure, whose output is a linear order extending $(S, \prec)$ if one exists, and which detects a cycle otherwise. The algorithm maintains two queues $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$, which store the elements of $S_\prec$ (respectively, $S_\succ$) for which we have not yet found all predecessors (respectively, successors). The procedure ENQUEUE adds an element to a queue. Similarly, DEQUEUE deletes an element from the queue. An element a is deleted from the data structure $\mathcal{D}_\prec$ by calling DELETE(a, $\mathcal{D}_\prec$); a deletion from $\mathcal{D}_\succ$ is performed with a similar call. To delete all elements in a set A, we simply write DELETE(A, $\mathcal{D}_\prec$).

The two main steps in the algorithm are steps 4 and 5. In step 4 of the algorithm the tandem search is performed; step 4(i) computes a new element of $S_\prec$, and step 4(ii) computes a new element of $S_\succ$. In step 5 the two sets that result from the partitioning are sorted recursively; to this end we first construct the data structures that are needed in the recursive calls. For the larger of the two sets the new data structures are obtained from the existing data structures, and for the smaller set the data structures are built from scratch.

```
ORDER(S,$\mathcal{D}_\prec$, $\mathcal{D}_\succ$)
1. **if** $|S| > 1$ **then** perform steps 2–6 **else** stop ($S$ is already sorted).
2. Make $S_\prec \leftarrow \emptyset$ and $S_\succ \leftarrow \emptyset$, and initialize two empty queues $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$.
3. Pick an arbitrary pivot element $a_{\text{piv}} \in S$; ENQUEUE($a_{\text{piv}}$, $\mathcal{Q}_\prec$); ENQUEUE($a_{\text{piv}}$, $\mathcal{Q}_\succ$).
4. **while** both $\mathcal{Q}_\prec$ and $\mathcal{Q}_\succ$ are nonempty
   **do** (i) $a \leftarrow$ DEQUEUE($\mathcal{Q}_\prec$); $a' \leftarrow$ QUERY($a$, $\mathcal{D}_\prec$).
          **if** $a' \neq NIL$
          **then if** $a' = a_{\text{piv}}$
                 **then** Stop and report that there is a cycle.
                 **else** ENQUEUE($a$, $\mathcal{Q}_\prec$) ; ENQUEUE($a'$, $\mathcal{Q}_\prec$); DELETE($a'$, $\mathcal{D}_\prec$).
                      $S_\prec \leftarrow S_\prec \cup \{a'\}$.
       (ii) Compute a new element $a' \in S_\succ$ in a similar way, using $\mathcal{Q}_\succ$ and $\mathcal{D}_\succ$.
5. **if** $\mathcal{Q}_\prec$ is empty (hence, $S_\prec$ is the smaller set)
   **then** Reinsert the elements of $S_\succ$ into $\mathcal{D}_\succ$.
        DELETE($S_\prec \cup \{a_{\text{piv}}\}$, $\mathcal{D}_\succ$); DELETE($a_{\text{piv}}$, $\mathcal{D}_\prec$).
        Build new predecessor and successor structures $\mathcal{D}'_\prec$ and $\mathcal{D}'_\succ$ for the set $S_\prec$.
        ORDER($S_\prec$, $\mathcal{D}'_\prec$, $\mathcal{D}'_\succ$).
        ORDER($S - \{a_{\text{piv}}\} - S_\prec$, $\mathcal{D}_\prec$, $\mathcal{D}_\succ$).
   **else** Compute the data structures for the recursive calls as above, reversing the
        roles of $S_\prec$, $\mathcal{D}_\prec$ and $S_\succ$, $\mathcal{D}_\succ$, and sort $S_\succ$ and $S - S_\succ - \{a_{\text{piv}}\}$ recursively.
6. Concatenate $S_\prec$, $a_{\text{piv}}$, and $S_\succ$ to form the ordered list for $S$.
```

The following lemma proves the correctness of our algorithm.

LEMMA 2.1. *Procedure* ORDER *outputs a linear order extending* $(S, \prec)$ *if it exists and detects a cycle otherwise.*

Proof. It is straightforward to see that the algorithm never claims to have found a cycle that does not exist. It remains to show that if ORDER outputs a list $a_1, \ldots, a_n$ then this list is a correct ordering. Assume for a contradiction that $a_i \succ a_j$ for some $i < j$. Then, at some stage of the algorithm, a_i must have been put into $S_\prec$, whereas a_j was put into $S_\succ$, or a_i was put into $S_\prec$ and a_j was the pivot element a_{piv}, or a_i was the pivot element a_{piv} and a_j was put into $S_\succ$. The second and third case both imply that there is a cycle containing a_{piv}, and we can easily verify that step 3 never fails to discover a cycle containing the pivot element. We thus consider the first case: If $\mathcal{Q}_\prec$ is empty after step 3 then all predecessors of a_i have been found, including a_j. Hence, a_j would have been put into $S_\prec$ instead of $S_\succ$. (It may also happen that a_j is put into both sets, but in that case the algorithm would have reported a cycle containing the pivot element.) Similarly, if $\mathcal{Q}_\succ$ is empty then a_i would have been put into $S_\succ$. □

Next we prove a bound on the running time of the algorithm. Let us for the sake of simplicity assume that the query time of $\mathcal{D}_\prec$ and the query time of $\mathcal{D}_\succ$ are equal, and let this time be denoted by $Q(n)$. Similarly, let the time to build these structures on n elements be $B(n)$, and let $D(n)$ denote the time for a deletion.

LEMMA 2.2. *The procedure* ORDER *runs in* $O\left([B(n) + n(Q(n) + D(n))]\log n\right)$ *time. The running time reduces to* $O\left(B(n) + n[Q(n) + D(n)]\right)$ *if* $B(n)/n + Q(n) + D(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Proof. Since all other operations in the procedure can be done in constant time, the time that we spend is dominated by the operations on the structures $\mathcal{D}_\prec$ and $\mathcal{D}_\succ$. Furthermore, if the size of the smaller of the two subsets $S_\prec$ and $S_\succ$ is m, then we perform at most $2m + 2$ queries and deletions on these structures in step 4 of the procedure. Restoring a data structure to a situation from the past, which we do in step 5, can be done without extra asymptotic overhead if we record all the changes. Finally, we perform m deletions in step 5, and we build new data structures for the smaller set. This adds up to $B(m) + O(1 + m)[Q(n) + D(n)]$ in total for the partitioning.

Next we argue that $m \leq n/2$ if the partitioning is successful, that is, if no cycle is found at this point. Suppose that $m > n/2$. Then there must be an element $a \in S_\prec \cap S_\succ$. But this means that a_{piv} will be found as a predecessor or a successor (whichever happens first) and a cycle is detected. Trivially, an unsuccessful partitioning happens at most once, giving a one-time cost of $O(n[Q(n) + D(n)])$.

It follows that the total running time $T(n)$ can be bounded by the recursion

$$T(n) \leq \max_{0 \leq m \leq n/2} \{B(m) + O(1 + m)[Q(n) + D(n)] + T(m) + T(n - m - 1)\},$$

which solves to the claimed time. □

Combining the two lemmas above, we obtain the following theorem.

THEOREM 2.3. *The procedure* ORDER *runs in* $O\left([B(n) + n(Q(n) + D(n))]\log n\right)$ *time and outputs an ordered list if* $(S, \prec)$ *does not contain a cycle or finds a cycle otherwise. The running time reduces to* $O\left(B(n) + n[Q(n) + D(n)]\right)$ *if* $B(n)/n + Q(n) + D(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Remark. With a little extra effort, the algorithm can output a witness cycle, when $(S, \prec)$ cannot be ordered. To this end, we keep track of the successor (predecessor) of each element that we put into $S_\prec$ ($S_\succ$). This extra information enables us to "walk back" when we find a_{piv} in step 3 of the algorithm and report the elements of the cycle.

3. Verifying linear extensions. In this section it is shown how to verify a given order for a relation $(S, \prec)$. Notice that different orders can be valid for $(S, \prec)$, so it does not suffice to compute a valid order and compare it to the given order. The algorithm uses a straightforward divide-and-conquer approach. It relies on the existence of a data structure $\mathcal{D}_\prec$ for predecessor queries. Unlike in the previous section, however, this data structure need not be dynamic. The algorithm we describe next has a list $\mathcal{L} = \{a_1, \ldots, a_n\}$ as input. For this list we have to test whether it corresponds to a valid order. It will report that $\mathcal{L}$ is not sorted or run quietly when $\mathcal{L}$ is a valid ordering for $(S, \prec)$.

```
VERIFY($\mathcal{L}$)
if $|\mathcal{L}| > 1$
then Let $\mathcal{L}_1 = \{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$ and $\mathcal{L}_2 = \{a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n\}$.
     Build a data structure $\mathcal{D}_\prec$ for predecessor queries on $\mathcal{L}_2$.
     for $i = 1$ to $\lfloor n/2 \rfloor$
     do if QUERY($a_i, \mathcal{D}_\prec$)$\neq$ NIL
        then Stop and report that $\mathcal{L}$ is not sorted
     VERIFY($\mathcal{L}_1$); VERIFY($\mathcal{L}_2$)
```

The correctness of the procedure is proved as follows. If $\mathcal{L}$ does not correspond to a valid order, then, by definition, there are elements a_i, a_j such that $a_i \prec a_j$ and $i > j$. Now either $i > \lfloor n/2 \rfloor$ and $j \leq \lfloor n/2 \rfloor$, or $i, j \leq \lfloor n/2 \rfloor$, or $i, j > \lfloor n/2 \rfloor$. The first case is tested by querying with the elements of $\mathcal{L}_1$ in the data structure $\mathcal{D}_\prec$, and the second and third possibilities are tested with the recursive calls for $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. The following theorem is now straightforward. As before, $B(n)$ denotes the time needed to build the structure $\mathcal{D}_\prec$ on a set of n elements, and $Q(n)$ denotes the query time.

THEOREM 3.1. *The procedure* VERIFY *verifies in* $O([B(n) + nQ(n)] \log n)$ *time whether a list* $\mathcal{L}$ *corresponds to an order for* $(S, \prec)$. *The running time of the procedure reduces to* $O\,(B(n) + nQ(n))$ *if* $B(n)/n + Q(n) = \Omega(n^\alpha)$ *for some constant* $\alpha > 0$.

Remark. Observe that if the procedure reports that $\mathcal{L}$ is not ordered, then it can report a witness pair a_i, a_j of elements such that $i < j$ and $a_j \prec a_i$. If the structure $\mathcal{D}_\prec$ is dynamic, then the algorithm can even report all conflicting pairs. When we test an element $a_i \in \mathcal{L}_1$, we just remove each element $a_j \in \mathcal{L}_2$ that conflicts with a_i from $\mathcal{D}_\prec$ and report the pair a_i, a_j, until no more conflicting elements are found. Then we reinsert the elements of $\mathcal{L}_2$ into $\mathcal{D}_\prec$ and test the next element of $\mathcal{L}_1$ in the same way.

4. Application to depth orders.

4.1. Depth orders for line segments. Let S be a set of n line segments in 3-space, and let $\vec{d}$ be the viewing direction. (The adaptation of the algorithms to "perspective depth orders," that is, depth orders with respect to a point, is straightforward.) We want to find a depth order on S for direction $\vec{d}$. In other words, we want to find a linear order extending the relation $(S, \prec)$, where $a \prec a'$ when there is a ray into direction $\vec{d}$ that first intersects line segment a' and then intersects line segment a. When $a \prec a'$, we say that a lies *behind* a' or that a' lies *in front of* a. Observe that $\prec$ is not necessarily a transitive relation. To apply Theorem 2.3, we need dynamic data structures that store a set $S' \subset S$ of line segments and enable us to find a line segment in S' lying behind (respectively, in front of) a query line segment. Define the *curtain* of a line segment into direction $\vec{d}$ to be the set of points q in 3-space such that there is a ray into direction $\vec{d}$ that first intersects a and then intersects q. If we want to find a line segment in S' lying in front of a query line segment a, we just have to check whether a intersects one of the curtains hanging from the line segments in S' and report the line segment holding that curtain. See Fig. 2. Finding a line segment lying behind a query line segment can be done in a similar way. Agarwal and Matoušek [2] have shown that intersection queries in a set of n curtains can be answered in time $O(n^{1/3})$ with a structure that uses $O(n^{4/3+\varepsilon})$ space and has an amortized update time of $O(n^{1/3+\varepsilon})$. (Note that the fact that the update time is amortized does not cause any difficulties for the analysis of the time bound.) If both the line segments holding the curtains and the query line segments are c-oriented, that is, they have only c different orientations for some constant c, then queries can be answered in time

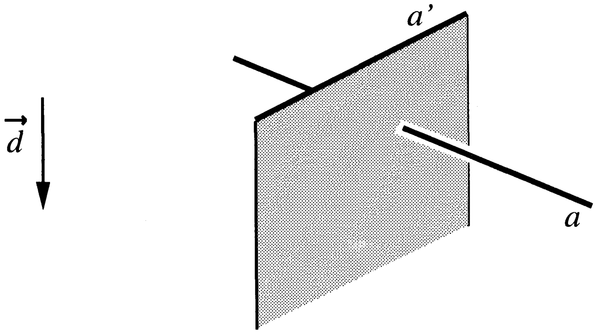


FIG. 2. *Line segment a lies behind line segment a′ and, hence, intersects its curtain.*

$O(\log^2 n)$ with a structure using $O(n \log n)$ space and with $O(\log^2 n)$ update time; see de Berg [7]. Combining this with Theorem 2.3 gives us the following result.

THEOREM 4.1. *Given a set S of n line segments in 3-space and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the line segments, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the line segments are c-oriented then the time bound improves to $O(n \log^3 n)$.*

To verify a given depth order for a set of line segments in 3-space, we use the results of §3. In the general case, we again use Agarwal and Matoušek's structure for predecessor and successor queries. In the c-oriented case, we can use a more efficient structure than we used for computing depth orders: because the structure need not be dynamic, we can use the structure of de Berg and Overmars [9], which has $O(\log n)$ query time and $O(n \log n)$ preprocessing time. We immediately obtain the following theorem.

THEOREM 4.2. *It is possible to verify a depth order on a given set S of n line segments in 3-space for a viewing direction $\vec{d}$ in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the line segments are c-oriented then the time bound improves to $O(n \log^2 n)$.*

4.2. Depth orders for triangles. To extend our results to triangles instead of line segments, we only need to adapt the data structures for predecessor and successor queries. Let us discuss the structure for successor queries; to obtain a structure for predecessor queries we only have to reverse the roles of "behind" and "in front of."

A triangle t is in front of another triangle t' if and only if at least one of the following conditions holds: (i) an edge of t is in front of an edge of t', (ii) t is in front of a vertex of t', or (iii) a vertex of t is in front of t'. (A vertex v is in front of a triangle t' if there is a ray into the viewing direction that first intersects v and then intersects t'.) We already know how to find the triangles t' that satisfy condition (i) for a query triangle t. The triangles satisfying conditions (ii) and (iii) can be found as follows. Consider condition (ii) and assume, to simplify the description, that the viewing direction is the negative z-direction. Project all vertices onto the xy-plane, and let $\bar{t}$ be the projection onto the xy-plane of a query triangle t. To find a vertex in front of t we select all vertices whose projections are contained in $\bar{t}$ in a small number of groups; for such a group we can think of t as being a plane, and the question becomes that of reporting a point in a half-space in 3-space. The latter query can be answered in $O(\log n)$ time, using the half-space emptiness structure of Agarwal et al. [1]. This structure uses $O(n^{1+\varepsilon})$ preprocessing and has $O(n^{\varepsilon})$ update time. The selection can be done using a three-level partition tree: each level filters out those vertices lying on the appropriate side of the line through one of the three edges of $\bar{t}$. We use the partition tree of Matoušek [17], which allows for queries and updates in time $O(n^{1/3})$ (respectively, $O(n^{1/3+\varepsilon})$), and which uses $O(n^{4/3+\varepsilon})$ preprocessing time and space. Because the preprocessing and the query times in such a multi-level partition tree are essentially determined by the least efficient level, the preprocessing time, query time, and update time of the total structure remain $O(n^{4/3+\varepsilon})$, $O(n^{1/3})$, and $O(n^{1/3+\varepsilon})$, respectively. See [2], [3], [6], [17] for further details on the analysis of multilevel partition trees. The structure for condition (iii) is the same (up to some dualizations) as the structure for (ii) that we just described. We conclude that a dynamic structure for predecessor queries in a set of triangles exists with $O(n^{1/3})$ query and $O(n^{1/3+\varepsilon})$ update time, using $O(n^{4/3+\varepsilon})$ preprocessing time and space.

For the c-oriented case, where the edges of the triangles have only c different orientations for some constant c, we can use structures from [7]. There it is shown that a vertex in front of a c-oriented query triangle can be found in $O(\log^3 n)$ time, with a structure that uses $O(n \log^2 n)$ space and has $O(\log^3 n)$ update time. A triangle in front of a query vertex can be found in $O(\log^2 n \log\log n)$ time, using $O(n \log^2 n)$ space and with $O(\log^2 n \log\log n)$ update time. Furthermore, finding a vertex in front of an axis-parallel rectangle can be done with a structure whose query and update time are $O(\log^2 n)$.

The above combined with Theorems 2.3 and 3.1 lead to the following result.

THEOREM 4.3. *Given a set S of n triangles in 3-space and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the triangles, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the triangles are c-oriented then the time bound improves to $O(n \log^4 n)$, and if the objects are axis-parallel rectangles then the algorithm takes $O(n \log^3 n \log\log n)$ time.*

To verify a given depth order for an arbitrary set of triangles, we use the same structures as for computing a depth order. In the c-oriented case, however, we can save some logarithmic factors by using static structures instead of dynamic ones. In the algorithm of §3 we have to test whether the triangles in a list $\mathcal{L}_1$ do not lie in front of any triangle in a list $\mathcal{L}_2$. Testing whether there is an edge of a triangle in $\mathcal{L}_1$ that lies in front of an edge of a triangle in $\mathcal{L}_2$ can be done in $O(n \log n)$ time, as in §4.1. To test for conflicts corresponding to conditions (ii), we build a structure on the triangles in $\mathcal{L}_1$ that reports the first triangle that is hit by a query ray starting from infinity into the viewing direction. Next, we shoot rays from infinity into the viewing direction toward each vertex of all triangles in $\mathcal{L}_2$; when we know the first triangle that is hit by the ray toward a certain vertex, we can decide if there is any triangle in front of the vertex. There exists a structure that answers these ray shooting queries in $O(\log n)$ time, after $O(n \log n)$ preprocessing [9]. Hence, in $O(n \log n)$ time we can decide if there is a triangle in $\mathcal{L}_1$ that is in front of some vertex of a triangle in $\mathcal{L}_2$. To test condition (iii) we build a similar structure on the triangles in $\mathcal{L}_2$ (only this time for query rays into the opposite viewing direction), and we query with vertices of triangles in $\mathcal{L}_1$. This leads to the following theorem.

THEOREM 4.4. *It is possible to verify a given depth order on a set S of n triangles in 3-space for a direction $\vec{d}$ in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the triangles are c-oriented then the time bound improves to $O(n \log^2 n)$.*

4.3. Extension to polygons. Consider the case where we want to compute a depth order for a set of polygons in 3-space, instead of a set of triangles. Let n be the total number of vertices of the polygons. First, we triangulate every polygon, which can be done in $O(n)$ time in total [4]. Observe that one polygon is behind another polygon if and only if one of the triangles in the triangulation of the first polygon is behind one of the triangles of the second polygon. Hence, we can use the same data structures as before to find predecessors and successors. However, if the polygons do not have constant complexity, then there is a slight problem; the triangles that correspond to the same polygon must stay together in the ordering, so when we find one triangle as a predecessor or successor we have to report the other triangles as well. This is problematic, because the number of other triangles can be large. Suppose that during our tandem search we suddenly have to add a very large polygon to one of the subsets; if we find out in the next step that the other subset is complete, then we have spent a lot of time that we cannot charge to the smaller subset. An elegant solution to this problem can be obtained if we realize that we can choose any particular pivot element we like. Hence, we can choose the polygon with the largest complexity as pivot element. The tandem search for the sets $S_{\prec}$ and $S_{\succ}$ now proceeds as follows. We find successors and predecessors using the data structures for triangles. However, when we find a large polygon for, say, $S_{\prec}$, we first allow $S_{\succ}$ to catch up. Thus we search for successors until the complexity of $S_{\succ}$—that is, the total number of vertices of all polygons in $S_{\succ}$—is greater than the complexity of $S_{\prec}$. When this happens, we start querying for predecessors again, and so forth, until one of the subsets is completed. This way the extra work that we have to do, caused by adding a large polygon to what turns out to be the larger set, is bounded by the time spent on one polygon. Since the pivot polygon is chosen to be the largest polygon in the set, we can charge this extra work to the pivot polygon. Clearly, each polygon is charged at most once this way, because in the recursive calls we do not consider the pivot element anymore. Thus the asymptotic running time of the algorithm remains the same, and we have the following theorem.

THEOREM 4.5. *Given a set S of polygons in 3-space with n vertices in total and a viewing direction $\vec{d}$, one can compute a depth order on S for direction $\vec{d}$ or decide that there is cyclic overlap among the polygons, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the polygons are c-oriented then the time bound improves to $O(n \log^4 n)$, and if the polygons are axis-parallel then the algorithm takes $O(n \log^3 n \log\log n)$ time.*

The adaptation of the verification procedure to polygons is fairly straightforward, and we leave it as an (easy) exercise to the reader.

THEOREM 4.6. *It is possible to verify a given depth order on a set S of polygons in 3-space with n vertices in total, for a viewing direction $\vec{d}$, in time $O(n^{4/3+\varepsilon})$, for any fixed $\varepsilon > 0$. If the polygons are c-oriented then the time bound improves to $O(n \log^2 n)$.*

5. Concluding remarks. We have shown that it is possible to compute a depth order for a set of line segments in 3-space in subquadratic time. More specifically, a depth order can be computed in $O(n^{4/3+\varepsilon})$ time in the general case and in $O(n \log^3 n)$ time in the c-oriented case. It is also possible to verify a given depth order, and the results can be extended to polygons instead of line segments. Our algorithms are based on a general framework to compute or verify a linear order extending an implicitly defined binary relation, which might have other applications as well.

When a depth order is needed as input to a hidden surface removal algorithm, we are not done if we detect a cycle: the cycles should be removed by cutting the objects into smaller pieces. Moreover, we would like to use as few cuts as possible. As mentioned in the introduction, binary space partitions are a way of cutting the objects to obtain a depth order, but there is no guarantee that the number of pieces in this scheme is small [20]. We leave the computation of the minimum (or a small) number of cuts as an open problem. See [5] for an initial study of these problems.

REFERENCES

[1] P. K. AGARWAL, D. EPPSTEIN, AND J. MATOUŠEK, *Dynamic half-space reporting, geometric optimization, and minimum spanning trees*, Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, 1992, pp. 80–89.

[2] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, Proc. 24th Annual ACM Symposium Theory Comput., 1992, pp. 517–526.

[3] P. K. Agarwal and M. Sharir, *Applications of a new space partitioning scheme*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[4] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[5] B. Chazelle, H. Edelsbrunner, L. J. Guibas, R. Pollack, R. Seidel, M. Sharir, and J. Snoeyink, *Counting and cutting cycles of lines and rods in space*, Comput. Geom. Theory Appl., 1 (1992), pp. 305–323.

[6] B. Chazelle, M. Sharir, and E. Welzl, *Quasi-optimal upper bounds for simplex range searching and new zone theorems*, Proc. 6th Annual ACM Symposium Comput. Geom., 1990, pp. 23–33.

[7] M. de Berg, *Dynamic output-sensitive hidden surface removal for c-oriented polyhedra*, Comput. Geom. Theory Appl., 2 (1992), pp. 119–140.

[8] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld, *Efficient ray shooting and hidden surface removal*, Proc. 7th Annual ACM Symposium Comput. Geom., 1991, pp. 21–30.

[9] M. de Berg and M. Overmars, *Hidden surface removal for c-oriented polyhedra*, Comput. Geom. Theory Appl., 1 (1992), pp. 247–268.

[10] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.

[11] H. Fuchs, Z. M. Kedem, and B. Naylor, *On visible surface generation by a priori tree structures*, Comput. Graph., 14 (1980), pp. 124–133.

[12] L. Guibas, M. Overmars, and M. Sharir, *Ray shooting, implicit point location, and related queries in arrangements of segments*, Report 433, Department of Computer Science, New York University, New York, NY, March 1989.

[13] L. J. Guibas and F. F. Yao, *On translating a set of rectangles*, Computational Geometry, F. P. Preparata, ed., Advances in Computing Research 1, JAI Press, London, England, 1983, pp. 61–77.

[14] R. H. Güting and T. Ottmann, *New algorithms for special cases of the hidden line elimination problem*, Comput. Vision Graph. Image Process., 40 (1987), pp. 188–204.

[15] C. Kenyon-Mathieu and V. King, *Verifying partial orders*, Proc. 21st Annual ACM Symposium Theory Comput., 1989, pp. 367–374.

[16] M. J. Katz, M. H. Overmars, and M. Sharir, *Efficient hidden surface removal for objects with small union size*, Comput. Geom. Theory Appl., 2 (1992), pp. 223–234.

[17] J. Matoušek, *Efficient partition trees*, Proc. 7th Annual ACM Symposium Comput. Geom., 1991, pp. 1–9.

[18] O. Nurmi, *On translating a set of objects in two- and three-dimensional spaces*, Comput. Vision Graph. Image Process., 36 (1986), pp. 42–52.

[19] M. Overmars and M. Sharir, *Output-sensitive hidden surface removal*, Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 598–603.

[20] M. S. Paterson and F. F. Yao, *Efficient binary space partitions for hidden-surface removal and solid modeling*, Discrete Comput. Geom., 5 (1990), pp. 485–503.

[21] F. P. Preparata, J. S. Vitter, and M. Yvinec, *Output-sensitive generation of the perspective view of isothetic parallelepipeds*, Algorithmica, 8 (1992), pp. 257–283.

[22] G. Toussaint, *Some collision avoidance problems between spheres*, Proc. International Conf. Systems, Man, Cybernetics, 1985.

[23] F. F. Yao, *On the priority approach to hidden-surface algorithms*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 301–307.

SIAM J. COMPUT.
Vol. 23, No. 2, pp. 447–448, April 1994

016

ERRATUM: OPTIMAL PARALLEL RANDOMIZED ALGORITHMS FOR THREE-DIMENSIONAL CONVEX HULLS AND RELATED PROBLEMS*

JOHN H. REIF† AND SANDEEP SEN‡

A portion of the appendix to "Optimal Parallel Randomized Algorithms for Three-Dimensional Convex Hulls and Related Problems," by John H. Reif and Sandeep Sen [SIAM J. Comput., 21 (1992), pp. 466–485] was inadvertently deleted when the article was printed. The appendix appears here in its entirety.

A. Appendix. We say a random variable X upper-bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, $\text{Prob}(X \leq x) \leq \text{Prob}(Y \leq x)$.

A Bernoulli trial is an experiment with two possible outcomes, success and failure. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can be easily seen to be

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}.$$

The tail end of the binomial distribution can be bounded by *Chernoff* bounds. In particular the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob}(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np} \tag{1}$$

$$\text{Prob}(X \leq m) \leq \left(\frac{np}{m}\right)^m e^{-np+m} \tag{2}$$

$$\text{Prob}(X \leq (1-\epsilon)pn) \leq \exp(-\epsilon^2 np/2) \tag{3}$$

$$\text{Prob}(X \geq (1+\epsilon)np) \leq \exp(-\epsilon^2 np/3) \tag{4}$$

for all $0 < \epsilon < 1$. The last two bounds actually follow from the Chernoff bounds, which (for a discrete distribution) can be stated as

$$\text{Prob}[A \geq x] \leq z^{-x} G_A(z)$$

where $G_A(z)$ is the probability generating function.

To minimize the bound we substitute $z = z_o$, which minimizes the right side expression.

Proof. (Lemma 10): Consider the generalized Chebychev's inequality

$$\text{Prob}\{|X| \geq t\} \leq \frac{E(\phi(X))}{\phi(t)}.$$

Using $\phi(t) = t^{2k}$ and substituting $X - E[X]$ for X and setting t to be equal to μ, we get

$$\text{Prob}\{|X - E[X]| \geq E[X]\} \leq \frac{E[(X - E[X])^{2k}]}{E^{2k}[X]}.$$

*Received by the editors September 10, 1993; accepted for publication September 10, 1993.

†Department of Computer Science, Duke University, Durham, North Carolina, 27706.

‡Department of Computer Science, Duke University, Durham, North Carolina, 27706. Present address, Department of Computer Science and Engineering, Indian University of Technology, New Delhi, 110016, India.

Let us focus on the numerator. We shall show that it is $O(\mu^k)$ and the lemma follows. Since $E[X] = \sum_{i=1}^{n} E[X_i]$, we can write $(X - E[X])^{2k}$ as $(\sum_{i=1}^{n} X_i - E[X_i])^{2k}$. In the multinomial expansion, all the terms containing $X_i - E[X_i]$ (for any i) as a factor vanish because of the $2k$-way independence property.

There are $\binom{n}{c} \cdot \binom{2k-1}{c-1}$ terms that have c distinct non-unit product terms of the form $(X_j - E[X_j])^i$ such that $i > 0$ and $\sum i = 2k$. Also note that

$$\begin{aligned} E[(X_j - E[X_j])^i] &= (1-p)(-p)^i + p(1-p)^i \\ &\leq p^i + p \\ &= p(1 + p^{i-1}) \\ &\leq 2p. \end{aligned}$$

We can factor out p so that we can write the coefficient of n^c as $p^c \cdot f(k)$, where f is a function independent of n and can be absorbed in the big-O notation. From our observation about the first-order terms (which vanish), the maximum value of c is k. The numerator can be bound by the asymptotically dominating term $O(n^k \cdot p^k) = O(\mu^k)$. Since the denominator is μ^{2k}, the lemma follows. □

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 449–465, June 1994

TOP-BOTTOM ROUTING AROUND A RECTANGLE IS AS EASY AS COMPUTING PREFIX MINIMA*

OMER BERKMAN†, JOSEPH JÁJÁ‡, SRIDHAR KRISHNAMURTHY§, RAMAKRISHNA THURIMELLA¶, AND UZI VISHKIN||

Abstract. A new parallel algorithm for the prefix minima problem is presented for inputs drawn from the range of integers $[1..s]$. For an input of size n, it runs in $O(\log\log\log s)$ time and $O(n)$ work (which is optimal). A faster algorithm is presented for the special case $s = n$; it runs in $O(\log^* n)$ time with optimal work. Both algorithms are for the Priority concurrent-read concurrent-write parallel random access machine (CRCW PRAM). A possibly surprising outcome of this work is that, whenever the range of the input is restricted, the prefix minima problem can be solved significantly faster than the $\Omega(\log\log n)$ time lower bound in a decision model of parallel computation, as described by Valiant [SIAM J. Comput., 4 (1975), pp. 348–355].

The top-bottom routing problem, which is an important subproblem of routing wires around a rectangle in two layers, is also considered. It is established that, for parallel (and hence for serial) computation, the problem of top-bottom routing is no harder than the prefix minima problem with $s = n$, thus giving an $O(\log^* n)$ time optimal parallel algorithm for the top-bottom routing problem. This is one of the first nontrivial problems to be given an optimal parallel algorithm that runs in sublogarithmic time.

Key words. parallel algorithms, VLSI routing, prefix minima, PRAM algorithms

AMS subject classifications. 68Q10, 68Q25

1. Introduction.

The prefix minima problem is defined as follows. Given an array of integers $A = [a_1, a_2, \ldots, a_n]$ drawn from the range of integers $[1..s]$, find for each i, $1 \le i \le n$, the minimum over the prefix $a_1, a_2, \ldots, a_i$. We give two parallel algorithms for the problem: An $O(\log\log\log s)$ time algorithm using $n/\log\log\log s$ processors and $O(ns^\epsilon)$ space, and an $O(\log^* n)$[1] time algorithm using an optimal number of processors and linear space when the input is drawn from the range of integers $[1..n]$ (i.e., $s = n$).

We demonstrate the power of prefix minima by giving a nontrivial application—the top-bottom routing around a rectangle in two layers. The routing problem is defined as follows. We are given a rectangle $\mathcal{R}$ and a set of n two-terminal nets with each net having exactly one terminal on the top and one terminal on the bottom of $\mathcal{R}$. Find a routing around $\mathcal{R}$, in the standard two-layer model that uses the minimum number of total tracks. We give an $O(\log^* n)$ time algorithm for this routing problem using an optimal number of processors. Our time bound compares favorably with time bounds of previous algorithms. Additionally, the algorithm presented in this paper is quite elegant and relatively simple.

The main model of parallel computation that is used in this paper is the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM). We assume that if several processors attempt to write at the same memory location the processor with minimum

*Received by the editors August 6, 1991; accepted for publication (in revised form) December 7, 1992.

†Department of Computer Science, King's College London, The Strand, London WC2R 2LS, England. The work of this author was carried out in part at the University of Maryland Institute for Advanced Computer Studies and at Tel Aviv University, and was supported in part by National Science Foundation grant CCR-8906949.

‡University of Maryland Institute for Advanced Computer Studies and Department of Electrical Engineering, College Park, Maryland 20742. The work of this author was supported in part by National Science Foundation grant CCR-9103135 and National Science Foundation Engineering Research Center Program NSFD CDR 8803012.

§Department of Electrical Engineering, University of Maryland, College Park, Maryland 20742.

¶Department of Mathematics and Computer Science, University of Denver, Denver, Colorado 80208. The work of this author was carried out at the University of Maryland Institute for Advanced Computer Studies.

||University of Maryland Institute for Advanced Computer Studies and Department of Electrical Engineering, College Park, Maryland 20742, and Tel Aviv University, Tel Aviv, Israel 69978. The work of this author was supported in part by National Science Foundation grant CCR-8906949.

[1] $\log^* n = \text{minimum}\{i \mid \log^{(i)}(n) \le 1\}$ where $\log^{(i)}$ denotes the ith iterate of log (i.e., $\log^{(1)}(n) = \log n$ and $\log^{(i)}(n) = \log(\log^{(i-1)}(n))$ for $i \ge 2$.)

index succeeds (the so-called PRIORITY CRCW PRAM). An *optimal* parallel algorithm is an algorithm whose time-processor product matches the sequential complexity of the problem (which in this paper is always linear in the size of the input). An *ultimate* theoretical goal, that is, often unattainable, is to design constant-time optimal parallel algorithms. Seeking to get as close as possible to accomplishing this goal is an interesting research goal.

Valiant [12] gave an $\Omega(\log\log n)$ time lower bound for computing the maximum of n elements using n processors in a comparison model of parallel computation. The same lower bound applies for a PRAM assuming the input is drawn from a large domain (see [8] and [9]). Our result shows that this lower bound does not apply when the input range is restricted. Computing the parity of n bits needs $\Omega(\log n/\log\log n)$ time using a polynomial number of processors [2]. Since parity is such a simple problem, it is interesting to demonstrate that other interesting and nontrivial problems can be solved extremely fast. Recently, Berkman and Vishkin [4] have shown that several interesting functions could be computed in time that is significantly faster than the lower bound for parity, e.g., in inverse-Ackermann time. See also the survey [13]. The results of this paper also contribute in this direction. Furthermore, the top-bottom routing problem is one of the most involved problems for which a sublogarithmic optimal parallel algorithm presently exists.

The previous algorithms for the prefix minima problem run in $O(\log\log n)$ time using an optimal number of processors ([3] and [11]). This upper bound cannot be improved when the input range is unrestricted because of the lower-bound by Valiant. Our prefix minima algorithm has already been used by Matias and Viskin [7] as an important subroutine in designing an $O(\log^* n)$ time algorithm for generating a random permutation.

The top-bottom routing problem is one of the important subproblems encountered by very large scale integration (VLSI) circuit designers during the layout of a chip. This problem and its variations, arise in hierarchical approaches to VLSI layout. It should be emphasised, though, that the top-bottom routing is a restricted version of the problem of routing wires around multiple rectangles, which is the general routing problem. Indeed, the former problem can be considered a "very" simple case of the latter problem, which is itself NP-hard. Nevertheless, a study of the top-bottom routing problem provides new insights into the general problem and hence an efficient parallel algorithm for this problem could conceivably be a part of the software for computer-aided design tools.

While our algorithm is the first parallel algorithm, several authors have studied this problem in the sequential context. Baker [1], and Sarrafzadeh and Preparata [10] give a simple linear-time algorithm for this problem. However, the layout resulting from their algorithms is not optimal. On the other hand, the known algorithms that achieve optimality in the layout area for this problem are unsatisfactory for other reasons. LaPaugh's algorithm [6] is computationally expensive: it takes $O(n^3)$ time. The algorithm of Gonzalez and Lee [5], although it runs in $O(n)$ time, is quite involved.

2. Top-bottom routing.

2.1. Preliminaries and notation. Given a grid structure, let $\mathcal{R}$ be an $l \times h$ rectangle with sides on grid-line segments. Let $\mathcal{N}$ be $\{N_1, N_2, \ldots, N_s\}$. Each N_i (referred to as a **net**) consists of a pair of grid points (referred to as **terminals**) $N_i = \{a_{i_1}, a_{i_2}\}$, where terminal a_{i_1} (referred to as a **top terminal**) is on the top boundary of $\mathcal{R}$, and terminal a_{i_2} (referred to as a **bottom terminal**) is on the bottom boundary of $\mathcal{R}$. The *top-bottom routing problem* is to specify a path along grid-line segments, that connects for each N_i both its terminals. The layout (routing) should be such that the paths (wires) are outside $\mathcal{R}$ and should satisfy the constraints of a given routing model. We consider the *standard two-layer model*, where the horizontal segments of a wire are on one layer and the vertical segments are on the other. The segments of two different wires in any one layer are allowed no contact with one another, but wires running on different layers are allowed to cross without being connected. When a connection is desired at a grid point between a horizontal and a vertical segment of the same wire, it is achieved by an explicit connection (referred to as a *via*). The problem is to find such a routing around $\mathcal{R}$, that uses the minimum number of horizontal grid-line segments (referred to as **tracks**). There are two frequently used facts about routing we assume in this paper; the proofs of these facts are either straightforward or in [1], [6], [5]: (1) Minimizing the total number of horizontal tracks also minimizes the total area used by the layout, since each net increases the width of the layout by one unit. (2) There is an optimal routing such that each net uses exactly four bends or vias.

We elaborate on the definition of the problem. Assume the input is given as follows: two arrays $T = [t_1, t_2, \ldots, t_n]$ and $B = [b_1, b_2, \ldots, b_n]$ that contain net numbers as they appear at the top and bottom of $\mathcal{R}$ in left-to-right order. Without loss of generality (wlog), assume that $T = [t_1, t_2, \ldots, t_n] = [1, 2, \ldots, n]$ and B is a permutation of the numbers $1, 2, \ldots, n$. Note that any routing solution uses at least n horizontal tracks. Figure 1a demonstrates a solution for a problem with $n = 4$. The wire that leaves top-terminal 1 takes a vertical track upwards for one unit-length, connects to the horizontal track layer and takes a horizontal track left-bound for two units, and connects to the vertical track layer and takes a vertical track downwards for $h + 3$ units (recall that h is the height of the rectangle). A total of $h + 4$ horizontal tracks are used in the routing shown in Fig. 1a. It is easy to argue that in general $h + n$ is the minimal number of horizontal tracks possible for the routing problem. We ignore the horizontal tracks occupied by the rectangle and say that the number of horizontal tracks used by the above routing equals four.

A routing will be represented by two sets of edges: the *top* edges and the *bottom* .edges between nets in the array B (observe that array T is not needed). Each net in B is an endpoint of at most one top edge and at most one bottom edge.

- A **top edge** (b_k, b_l) means that nets b_k and b_l share the same track at the top and thus implies **directions** for b_k and b_l: if $b_k < b_l$, the direction of the net b_k is counter clockwise (CC) and that of b_l is clockwise (C); if $b_k > b_l$, the direction of b_k is C and that of b_l is CC. Observe that this is the direction as we traverse the path of the net from its top terminal to its bottom terminal.
- A **bottom edge** (b_i, b_j) means that b_i and b_j share the same track at the bottom and thus implies directions of b_i and b_j: if $i < j$, the direction of b_i is CC and that of b_j is C; if $i > j$, the direction of b_i is C and that of b_j is CC (see Fig. 1b).

A routing is *inconsistent with respect to* k, $1 \le k \le n$, if the direction implied by the top edge of a net b_k is C (respectively, CC) and the direction implied by the bottom edge of net b_k is CC (respectively, C); otherwise it is *consistent with respect to* k. A routing is **consistent** if it is consistent with respect to every k, $1 \le k \le n$.

An equivalent formulation of the consistency issue follows. Consider the complete undirected graph whose vertices are the nets. A **matching** in this graph consists of any subset of the edges such that no two of them share an endpoint. The set of top edges, denoted M_1, induces a matching; similarly the set of bottom edges, M_2, induces a matching. We define this pair of matchings M_1 and M_2 to be *consistent with respect to* a net b_i if, whenever $(b_i, b_j) \in M_1$ and $(b_i, b_k) \in M_2$, $b_i < b_j$ if and only if b_i appears before b_k in B (i.e., $i < k$). (M_1 and M_2 are also consistent with respect to b_i if b_i is unmatched in M_1 or in M_2.) A pair of matchings is called *consistent* if it is consistent with respect to each b_i. Our problem is to find a consistent pair of matchings M_1 and M_2 on B such that $|M_1| + |M_2|$ is a maximum.

Figure 1b shows an example of a consistent routing. For instance, for $b_2 = 2$, the top edge leads to $b_3 = 1$, implying a CC direction for net b_3 and a C direction for net b_2 (since $b_3 < b_2$). The bottom edge for b_2 leads to $b_1 = 3$, implying a CC direction for net b_1 and a C direction for net b_2 (since b_1 occurs before b_2 in B).

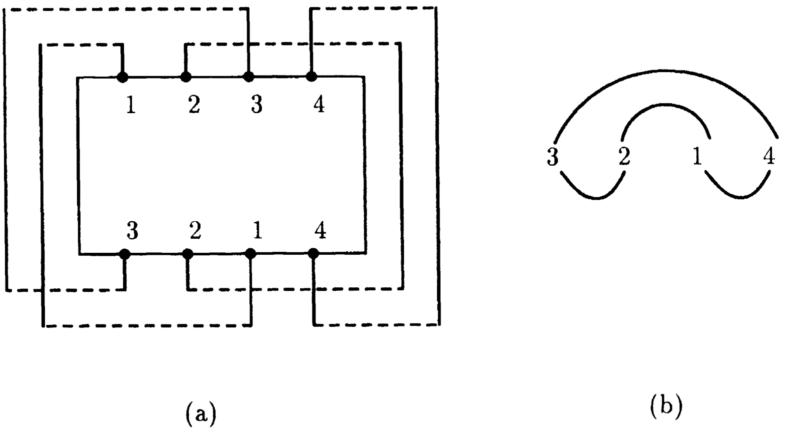


FIG. 1. *An example of top-bottom routing.* (a) *Actual routing.* (b) *Routing representation.*

Given such a representation with k_1 top edges and k_2 bottom edges, it is straightforward to assign tracks sequentially to the nets and thereby construct the actual routing. (Parallel track assignment is handled later.) The total number of tracks used at the top is $n - k_1$ and the total number of tracks used at the bottom is $n - k_2$. An optimal routing is thus a routing that maximizes $k_1 + k_2$.

2.2. An easy upper bound of n + 1 tracks. The proof of the following theorem provides an exceptionally simple algorithm. Its simplicity is particularly remarkable since the algorithm achieves the same number of tracks as in the pair up algorithms of Baker [1], and in Sarrafzadeh and Preparata [10], which are much more involved. Our method has the additional advantage that it can be implemented in constant time using n processors.

THEOREM 2.1. *A routing of the nets that uses a total of $n + 1$ tracks on the top and bottom parts of $\mathcal{R}$ can always be constructed.*

Proof. Let $B = [b_1, b_2, \ldots, b_n]$ be the bottom array. The set of top edges is given by (b_{2i}, b_{2i+1}), for $i = 1, 2, \ldots, \lfloor (n-1)/2 \rfloor$. The set of bottom edges is $(b_1, \max(b_2, b_3))$, $(\min(b_{2i}, b_{2i+1}), \max(b_{2i+2}, b_{2i+3}))$ for $i = 1, 2, \ldots, \lfloor (n-3)/2 \rfloor$. If n is an even integer, add also the bottom edge $(\min(b_{n-2}, b_{n-1}), b_n)$. When n is an odd integer, we have $(n-1)/2$ top edges and $(n-1)/2$ bottom edges. When n is even, we have $(n/2) - 1$ top edges and $n/2$ bottom edges. In both cases, the total number of tracks is $n + 1$.

It can be easily verified that the resulting routing is consistent. □

COROLLARY 2.2. *An $(n + 1)$-track routing solution can be obtained in sequential linear time and in $O(1)$ time with n processors on an exclusive-read exclusive-write* (EREW) PRAM.

Figure 2 gives examples of $(n + 1)$-track routing solutions.

2.3. Can we do better than n + 1? An $(n + 1)$-track routing solution always exists as shown above. It is easy to see that when n is odd this solution is optimal. Therefore, for the rest of the section assume n is even.

We describe a necessary and sufficient condition for the routing problem to have an n track solution.

DEFINITION 1. *A bottom array $B = \{b_1, b_2, \ldots, b_n\}$ is said to be* **unbalanced** *if there exists an odd integer i, such that the minimum over the prefix $\{b_1, b_2, \ldots, b_i\}$ is larger than the maximum over the suffix $\{b_{i+1}, b_{i+2}, \ldots, b_n\}$. (Alternatively, since B is a permutation of $1, 2, \ldots, n$ we can say that B is unbalanced if there exists an odd integer i such that the subarray $\{b_1, b_2, \ldots, b_i\}$ is a permutation of the numbers $\{n - i + 1, n - i + 2, \ldots, n\}$.) The array B is said to be* **balanced** *otherwise.*

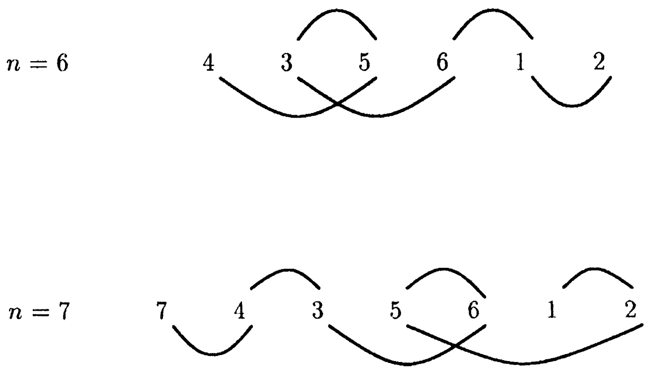


FIG. 2. $(n + 1)$*-track routing solutions.*

THEOREM 2.3. *Given an array B representing a top-bottom routing problem, an n track routing solution exists if and only if array B is balanced.*

Proof (*only if*). The proof is by contradiction. Assume B is unbalanced and let i be an odd integer such that the subarray $\{b_1, b_2, \ldots, b_i\}$ is a permutation of the numbers $\{n - i + 1, n - i + 2, \ldots, n\}$. Assume that the routing problem has an n track solution represented by $n/2$ top edges and $n/2$ bottom edges. At least $\lceil \frac{i}{2} \rceil$ nets from the subarray $\{b_1, b_2, \ldots, b_i\}$ must have a bottom edge with a net that lies to their right (in array B). On the other hand, at the top there exists a subset of nets $\{c_1, c_2, \ldots, c_{\lceil \frac{i}{2} \rceil}\} \subset \{n - i + 1, n - i + 2, \ldots, n\}$ such that each c_j, $j = 1, 2, \ldots, \lceil \frac{i}{2} \rceil$, shares a track with a net b_k where $b_k < c_j$. The direction implied for each of these $\lceil \frac{i}{2} \rceil$ nets is C. Since i is odd the direction assignments implied by these two sets of top and bottom edges results in a conflicting direction assignment for at least one net. This gives us the contradiction.

(*if*). Assume B is balanced. We construct an n track routing solution. We leave it to the reader to verify that the routing is consistent whenever we show how to assign directions to the nets. Denote by PMIN(i) the minimum over the prefix $\{b_1, b_2, \ldots, b_i\}$ and by SMAX(i) the maximum over the suffix $\{b_i, b_{i+1}, \ldots, b_n\}$ for $1 \leq i \leq n$.

We prove, inductively, that for an odd integer $i = 1, 3, \ldots, n - 1$, the sequence $\{b_1, b_2, \ldots, b_i, \text{SMAX}(i + 1)\}$ has an $i + 1$ track routing solution.

Basis. $i = 1$; Since B is balanced, $b_1 < \text{SMAX}(2)$ and a two track routing solution is given by the bottom edge $(b_1, \text{SMAX}(2))$ and the top edge $(b_1, \text{SMAX}(2))$.

Inductive step. Given an $(i - 1)$-track solution for $\{b_1, b_2, \ldots, b_{i-2}, \text{SMAX}(i - 1)\}$ we construct an $(i + 1)$-track solution for the sequence

$$\{b_1, b_2, \ldots, b_{i-2}, b_{i-1}, b_i, \text{SMAX}(i + 1)\}.$$

Case 1. $\text{SMAX}(i - 1) = \text{SMAX}(i + 1) = s$. We have a bottom edge (b_l, s) for some $l \leq i - 2$ in the inductively assumed $i - 1$ track solution. Replace this bottom edge by the following three edges: bottom edge $(b_l, \max(b_{i-1}, b_i))$, top edge (b_{i-1}, b_i), and bottom edge $(\min(b_{i-1}, b_i), s)$. The resulting set of top and bottom edges gives an $(i + 1)$-track solution for $\{b_1, b_2, \ldots, b_{i-2}, b_{i-1}, b_i, s\}$ (see Fig. 3).

Case 2. $\text{SMAX}(i - 1) \neq \text{SMAX}(i + 1)$. This implies that either $\text{SMAX}(i - 1) = b_{i-1}$ or $\text{SMAX}(i - 1) = b_i$. Assume $\text{SMAX}(i - 1) = b_{i-1}$ (the case $\text{SMAX}(i - 1) = b_i$ can be handled similarly). Let $p = \text{PMIN}(i)$. Since B is balanced, $p < \text{SMAX}(i + 1)$.

Case 2.1. $p = b_i$. We add a bottom edge $(b_i, \text{SMAX}(i+1))$ and a top edge $(b_i, \text{SMAX}(i+1))$, which gives an $(i + 1)$-track solution (see Fig. 4).

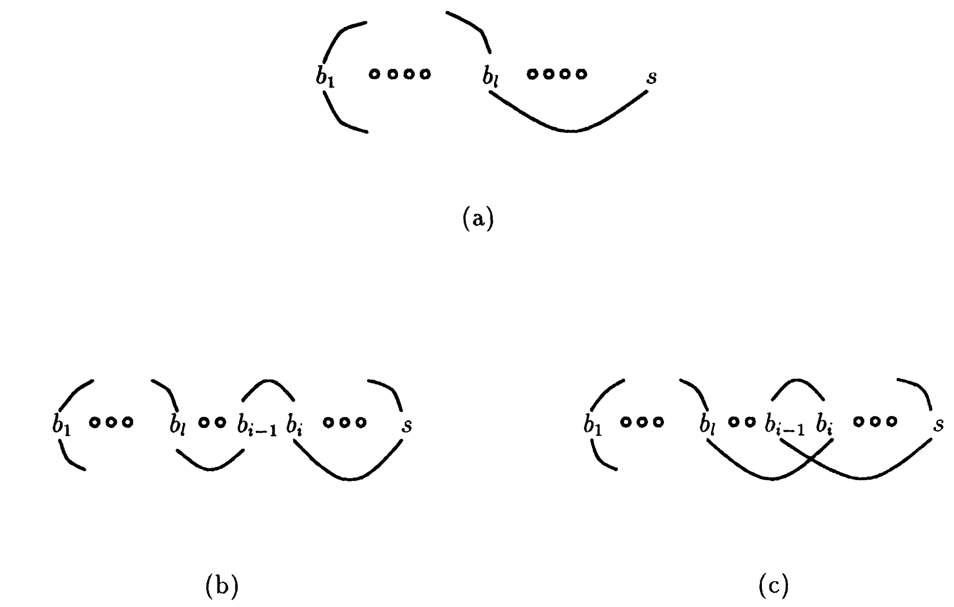

b_1 oooo b_l oooo s
(a)
b_1 ooo b_l oo b_{i-1} b_i ooo s
(b)
b_1 ooo b_l oo b_{i-1} b_i ooo s
(c)

FIG. 3. (a) *Inductively assumed* $(i-1)$*-track solution.* (b) $(i+1)$*-track solution for* $b_{i-1} > b_i$. (c) $(i+1)$*-track solution for* $b_{i-1}(i+1)$*-track solution for* $b_{i-1} < b_i$.

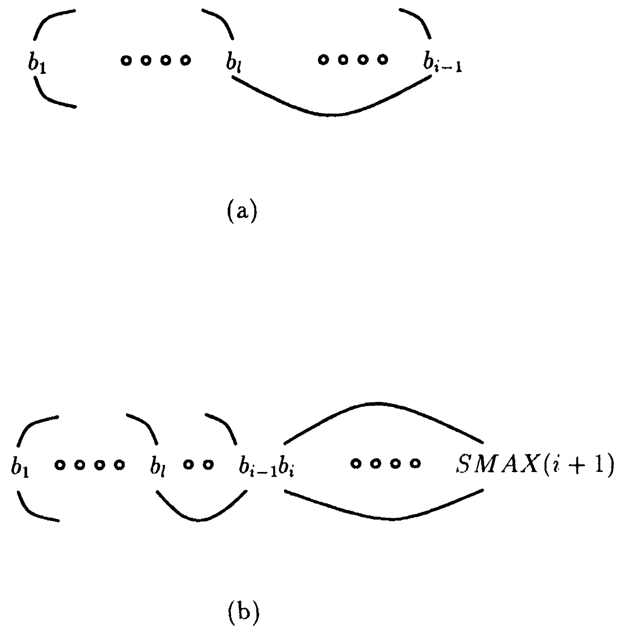
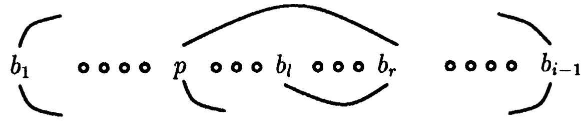

b_1 oooo b_l oooo b_{i-1}
(a)
b_1 oooo b_l oo $b_{i-1}b_i$ oooo $SMAX(i+1)$
(b)

FIG. 4. (a) *Inductively assumed* $(i-1)$*-track solution.* (b) $(i+1)$*-track solution.*

Case 2.2. $p \neq b_i$. We know that $b_{i-1} > b_i$ (since SMAX$(i-1) = b_{i-1}$). Thus $p \neq b_{i-1}$. It follows that there exists j, $1 \leq j \leq i-2$, such that $b_j = p$. We have a top edge (p, b_r), for some $r \leq i-1$ in the inductively assumed $i-1$ track solution. We also have a bottom edge

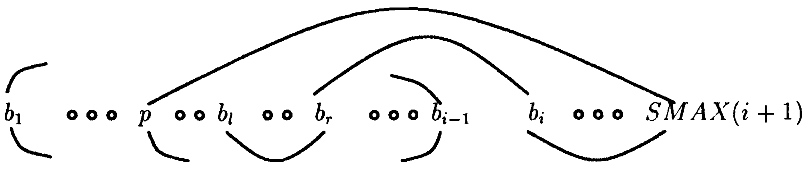
(b_l, b_r), for some $l \leq i-1$ (actually $l < r$ since the direction implied by the top edge (p, b_r) for net b_r is C).

Case 2.2.1. $b_r > b_i$. The top edge (p, b_r) is replaced by the following three edges: top edge $(p, \text{SMAX}(i+1))$, bottom edge $(\text{SMAX}(i+1), b_i)$, and top edge (b_i, b_r). This gives an $(i+1)$-track solution (see Fig. 5).



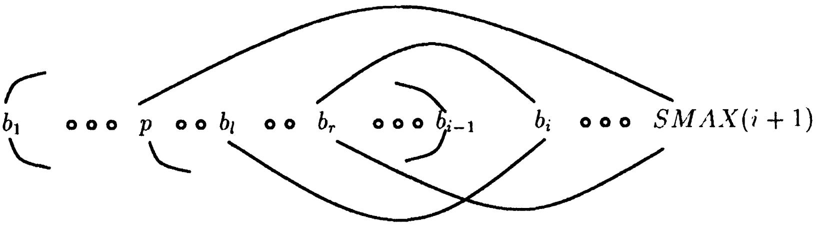
(a)

FIG. 5. (a) *Inductively assumed* $(i-1)$*-track solution.*



(b)

FIG. 5. (b) $(i+1)$*-track solution for* $b_i < b_r$.



(c)

FIG. 5. (c) $(i+1)$*-track solution for* $b_i > b_r$.

Case 2.2.2. $b_r < b_i$. We replace the top edge (p, b_r) and the bottom edge (b_r, b_l) by the following four edges: top edge $(p, \text{SMAX}(i+1))$, bottom edge $(\text{SMAX}(i+1), b_r)$, top edge (b_r, b_i), and bottom edge (b_i, b_l). Note that this is the only case in which the direction of any net is reversed with respect to its direction in the inductively assumed solution. This again gives an $(i+1)$-track solution (see Fig. 5). □

Given a bottom array B representing an instance of a top-bottom routing problem, we have the following theorem.

THEOREM 2.4. *An optimal routing solution can be constructed in sequential linear time.*

Proof. It is straightforward to verify whether array B is balanced. If B is unbalanced, then the optimal solution needs $n + 1$ tracks and Theorem 2.1 gives the construction of such a solution. Otherwise, the constructive proof of Theorem 2.3 gives an algorithm that finds an n track solution in linear time. □

2.4. Parallel algorithm. In this subsection we present a parallel version of the optimal routing algorithm discussed above.

Assume that the given input array $B = [b_1, b_2, \ldots, b_n]$ is balanced. To simplify the presentation of the algorithm we define an operation SWAP on array B. Given two indices i and j in the array, SWAP(i, j) swaps the contents of locations i and j in B. We show later, as a part of the correctness proof of our algorithm, that the directions assigned for the nets for the array after a SWAP operation can also be used for the array before the SWAP operation. First, we do the following preprocessing.

A balanced array $B = [b_1, b_2, \ldots, b_n]$ is said to satisfy the **even>odd property** if $b_{2i} > b_{2i+1}$ for all i, $1 \leq i < n/2$ in B. We use the SWAP operation, if necessary, to make the array satisfy the above property. That is, for every $1 \leq i < n/2$, if $b_{2i} < b_{2i+1}$, then SWAP$(2i, 2i + 1)$. For convenience, we continue to refer to the swapped array as B also.

Next find all prefix minima and suffix maxima with respect to B and put them in arrays PMIN and SMAX, respectively. For each odd integer i, if $b_i =$ PMIN(i) mark location i by "P." Similarly, for each even integer i, if $b_i =$ SMAX(i) mark location i by "S."

As a final part of the preprocessing step we further limit the input array to satisfy the **SP-free property**: for each even j, $2 \leq j < n$, either j is not marked S or $j + 1$ is not marked P. This is achieved by partitioning the input array as follows. Simultaneously, for every even integer j, if location j is marked "S" and location $j + 1$ is marked "P," then split array B between locations j and $j + 1$. These simultaneous splits partition array B into several contiguous subarrays.

CLAIM 2.5. *Each subarray after the above partition is even in length and is balanced. Furthermore, it satisfies*: (i) *the SP-free property, and* (ii) *the even > odd property.*

Proof. Each subarray after the partition is such that the first and last elements of the subarray are marked P and S, respectively. It is easy to see that each subarray satisfies the SP-free property and the even>odd property. Also, each subarray is of even length because markings S and P occur at an even and odd positions of the array, respectively. To prove that each subarray is balanced, we note that for each location in the subarray, its prefix minimum (resp. suffix maximum) with respect to the subarray is equal to its prefix minimum (resp. suffix maximum) with respect to B. Since B is balanced, it follows that the resulting subarrays are also balanced. □

Let $D = [d_1 \ldots d_m]$ be a subarray obtained after the preprocessing. The rest of the algorithm is devoted to solving the matching problem with respect to each such subarray separately. We note that array D is not a permutation of the numbers $1 \ldots m$. However, the crucial information about array D (namely, the marks "P" and "S") is already known and thus there is no need to compute prefix-minima or suffix-maxima with respect to D. We describe the algorithm below.

Algorithm *Parallel Top-Bottom Routing.*

Input: A balanced array D satisfying the even > odd property and the SP-free property. Also, the marks P and S for the elements of D.

Output: The directions C and CC for each of the elements of D.

1. (*Extract PS-pairs.*) Locate all pairs (d_l, d_r), $3 \leq l < r \leq m - 2$, such that d_l is marked "P," d_r is marked "S," and there is no element d_k, $l < k < r$, such that d_k is marked. Such a pair is called a **PS-pair**.

2. (*Handle PS-pairs.*) Consider first the case where there are no PS-pairs. For such a case add the top edge (d_1, d_m), and bottom edges (d_1, d_2) and (d_{m-1}, d_m) and proceed to Step 3. For the rest of this step we assume that there exists at least one PS-pair. We add the following edges:
   (a) (*Chain all outside neighbors of each PS-pair.*) For each PS-pair (d_l, d_r), add the top edge (d_{l-1}, d_{r+1}). Further, if $d_{l-1} < d_{r+1}$ then SWAP$(l - 1, r + 1)$. Add the bottom edges (d_{l-2}, d_{l-1}) and (d_{r+1}, d_{r+2}) (see Fig. 6a).
   (b) (*Chain all PS-pairs.*)
      (i) (*Attach at the extremes.*) Let (d_{l_1}, d_{r_1}) and (d_{l_2}, d_{r_2}) be the first and last PS-pairs in D. Add top edges (d_1, d_{r_1}) and (d_{l_2}, d_m). Add bottom edges (d_1, d_2) and (d_{m-1}, d_m) if no such edges exist from Step 2a (see Fig. 6b).
      (ii) (*Attach the middle pairs.*) For each two consecutive PS-pairs $(d_{l_\alpha}, d_{r_\alpha})$ and $(d_{l_\beta}, d_{r_\beta})$, add a top edge $(d_{l_\alpha}, d_{r_\beta})$. For each PS-pair (d_l, d_r), add bottom edges (d_l, d_{l+1}) and (d_{r-1}, d_r) (see Fig. 6c).
3. (*Pair-up the rest of the elements, each to its neighbor, based on their position: even to odd at the top and odd to even at the bottom.*) Add the following edges:
   Add top edges (d_{2j}, d_{2j+1}), $1 \leq j \leq (m/2) - 1$ unless either d_{2j} or d_{2j+1} have top edges from Step 2. Also add bottom edges (d_{2j-1}, d_{2j}), $2 \leq j \leq (m/2) - 1$ unless either d_{2j-1} or d_{2j} have bottom edges from Step 2.

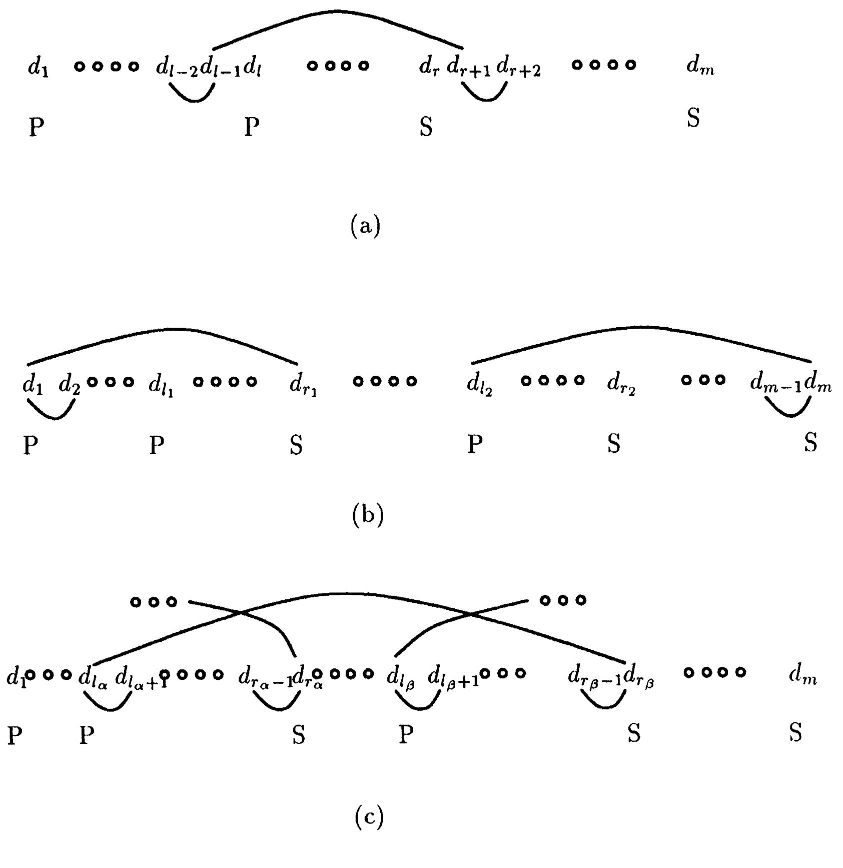

d_1 oooo $d_{l-2}d_{l-1}d_l$ oooo d_r d_{r+1} d_{r+2} oooo d_m
P P S S
(a)
d_1 d_2 ooo d_{l_1} oooo d_{r_1} oooo d_{l_2} oooo d_{r_2} ooo $d_{m-1}d_m$
P P S P S S
(b)
ooo ooo
d_1 ooo d_{l_α} $d_{l_\alpha+1}$ oooo $d_{r_\alpha-1}d_{r_\alpha}$ ooo d_{l_β} $d_{l_\beta+1}$ ooo $d_{r_\beta-1}d_{r_\beta}$ oooo d_m
P P S P S S
(c)

FIG. 6. *Handling of PS-pairs.*

Next, we argue the correctness of this algorithm. It follows from Step 3 that every element is paired up. We divide the remaining proof into two categories. First, we show that if an attempt is made by the algorithm to add an edge for an element of D at a particular stage, then that element is "available" to be paired up, i.e., no edge was added to that element of D by a step prior to the current one. The second part of the correctness argument consists of showing that the routing resulting from the algorithm is consistent (for the array before as well as after the SWAP operations).

2.4.1. Correctness I: no conflicts.

PROPOSITION 2.6. *Following the preprocessing, the outside neighbors d_{l-1} and d_{r+1} of any PS-pair (d_l, d_r) are unmarked (i.e., they are not marked neither "P" nor "S").*

Proof. Consider any PS-pair (d_l, d_r) and its outside neighbors d_{l-1} and d_{r+1}. Since $l-1$ is even, d_{l-1} cannot be marked "P." In addition since array D is SP-free, and d_l is marked "P," it implies that d_{l-1} cannot be marked "S." Similarly we can prove that d_{r+1} in unmarked. □

CLAIM 2.7. *The top and bottom edges added in Step* 2b(ii) *never conflict with an edge added in Step* 2a *(that is, if a top (bottom) edge (u, v) is added in Step* 2b(ii), *neither u nor v have top (bottom) edges from Step* 2a).

Proof. For top edges the claim follows directly from Proposition 2.6. Consider any PS-pair (d_l, d_r). The bottom edges that are added in Step 2a are (d_{l-2}, d_{l-1}) and (d_{r+1}, d_{r+2}). We prove the claim with respect to elements d_{l-2} and d_{l-1} (similar arguments work for d_{r+1} and d_{r+2}). We use three facts. (1) Elements d_{l-1} and d_{l-2} are not part of any PS-pairs. (2) At least one of the elements corresponding to a bottom edge added in Step 2b(ii) belong to a PS-pair. (3) All bottom edges that are added in Step 2b(ii) are of the form (d_{2j-1}, d_{2j}). Using fact (3) we conclude that the only possible edge that might have been added to d_{l-2} or d_{l-1} in Step 2b(ii) is (d_{l-2}, d_{l-1}). However, facts (1) and (2) together imply that such an edge cannot be added in Step 2b(ii). We are left to show why fact (1) is correct (facts 2 and 3 are obvious). Element d_{l-1} is unmarked (Proposition 2.6) and cannot be a part of any PS-pair. If element d_{l-2} is marked, it can only be marked "P." Thus the only way in which d_{l-2} can be a part of a PS-pair is if d_{l-1} is marked "S," which is not possible (Proposition 2.6). □

PROPOSITION 2.8. *Whenever an existing bottom edge in Step* 2b *and in Step* 3 *prevents us from connecting two elements u and v by a bottom edge, then both u and v are endpoints of existing bottom edges. Similarly in Step* 3, *if an existing top edge prevents us from connecting two elements u and v by a top edge, then both u and v are endpoints of existing top edges.*

Proof. We consider the top edges and the bottom edges separately.

(i) The proposition is trivial with respect to bottom edges: All bottom edges (u, v) are of the form (d_{2j-1}, d_{2j}) for some j, $1 \leq j \leq m/2$. Hence, in either Step 2b or Step 3, if we are prevented from connecting two elements u and v by a bottom edge, it implies that u and v are already connected to each other by an existing bottom edge.

(ii) All top edges (u, v) added in Step 3 are of the form (d_{2j}, d_{2j+1}), for some j, $1 \leq j \leq (m/2) - 1$. Prior to Step 3, whenever we add a top edge matching an element at i with an element at j, where i is even $(i < n)$, and j is odd $(j > 1)$, we also add top edges for the elements at $i + 1$ and $j - 1$ (this includes both cases $i < j$ and $i > j$). Hence if we are prevented from connecting two elements u and v by a top edge in Step 3, it implies that both u and v are endpoints of existing top edges. □

2.4.2. Correctness II: routing consistency.

The following proposition is useful for the later proofs.

PROPOSITION 2.9. *Given any two consecutive PS-pairs $(d_{l_\alpha}, d_{r_\alpha})$ and $(d_{l_\beta}, d_{r_\beta})$, we have $d_{l_\alpha} < d_{r_\beta}$. In addition, $d_1 < d_{r_1}$ and $d_{l_2} < d_m$. (If there are no PS-pairs then $d_1 < d_m$.)*

Proof. Consider the smallest index $i > r_\alpha$ that is marked P. The maximal element in the suffix $[i-1,\ldots,m]$ is d_{r_β}. The minimal element in the prefix $[1,\ldots,i-2]$ is d_{l_α}. (To see these, recall the SP-free property and the definition of PS-pairs.) Since D is balanced, $d_{l_\alpha} < d_{r_\beta}$. Similarly, we can prove that $d_1 < d_{r_1}$ and $d_{l_2} < d_m$. □

CLAIM 2.10. *The matchings corresponding to the top and bottom edges added in Step* 2a *are consistent with respect to d_{l-1} and d_{r+1}, for all PS-pairs (d_l, d_r) in D.*

Proof. We show that the matching is consistent with respect to d_{l-1}. Similar arguments work for d_{r+1}. From Step 2a, d_{l-1} has top edge (d_{l-1}, d_{r+1}) and bottom edge (d_{l-2}, d_{l-1}). Since $d_{l-1} > d_{r+1}$ (after the SWAP operation), the matchings corresponding to these edges are consistent in the swapped array D. It remains to prove that these matchings are consistent in array D before the SWAP of Step 2a. Consider the bottom edge (d_{l-2}, d_{l-1}). The elements at locations $l-2$ and $l-1$ could have changed their locations by this SWAP operation. Notice, however that all SWAP operations performed in this step are of the form SWAP(r, s) where r is even, s is odd and $r < s$. Since $l-1$ is even and $l-2$ is odd, it follows that element d_{l-1} must have been in location $\geq l-1$ and element d_{l-2} must have been in location $\leq l-2$ before the SWAP. Thus the matching is consistent with respect to d_{l-1} also before the SWAP. □

CLAIM 2.11. *The matchings resulting from the top and bottom edges added in Step* 2b *are consistent with respect to d_l and d_r for all PS-pairs (d_l, d_r) in D (and also with respect to d_1 and d_m).*

Proof. From Proposition 2.9 it is easy to see that the matchings are consistent with respect to the swapped array D. We show that neither d_l nor d_{l+1} are affected by the SWAP (of Step 2a). This will imply that the matchings are consistent also with respect to array D before the SWAP. Element d_l is unaffected by the SWAP operation since it is marked "P." For d_{l+1} to be affected d_{l+2} must be marked "P." If $l+1 = r$ then $l+2$ cannot be marked "P" because of the SP-free property. If $l+1 < r$ then $l+2$ cannot be marked "P" since otherwise (d_l, d_r) would not have been a PS-pair. Similarly, we can show that d_{r-1} and d_r are not affected by the SWAP operation. □

CLAIM 2.12. *The matchings resulting from the edges added in Step* 3 *are consistent with respect to both d_{2j} and d_{2j+1} for $1 \leq j \leq (m/2)-1$.*

Proof. For array D after the SWAP of Step 2a, the claim follows from the even > odd property. Since all SWAP operations of Step 2a are of the form SWAP(r, s) where r is even, s is odd, and $r < s$, it follows that element d_{2j} must have been in location $\geq 2j$ and d_{2j+1} must have been in location $\leq 2j+1$ before the SWAP operation. Thus the matchings are consistent also with respect to the original array D. □

MAIN CLAIM. *Steps* 1 *to* 5 *result in a consistent pair of matchings M_1 and M_2 for D, such that $|M_1| = |M_2| = m/2$.*

Proof. The Main Claim follows from Claims 2.10, 2.11, and 2.12. □

THEOREM 2.13. *M_1 and M_2 are the needed matchings on B (recall that B is the original array for which D is a subarray).*

Proof. From the Main Claim above, matchings M_1 and M_2 are consistent with respect to D and hence are also consistent with respect to array B obtained after the SWAP performed in the preprocessing step. Since our algorithm never puts a bottom edge matching the element at b_{2i} with the element at b_{2i+1}, for $1 \leq i < n/2$, matchings M_1 and M_2 are consistent also with respect to the original array B. □

2.4.3. Complexity and implementation.

THEOREM 2.14. *The optimal routing algorithm runs in $O(\log^* n)$ time using $O(n)$ operations on the* CRCW PRAM.

Proof. By Theorem 3.4 below we can solve the prefix-minima (or suffix-maxima) problem in $O(\log^* n)$ time and $O(n)$ operations. Remaining nontrivial implementation details are extracting PS-pairs (in Step 3), finding in each SP-free subarray the first and last PS-pairs (needed for Step 2b(i)), and finding consecutive PS-pairs (in Step 2b(ii)). Given the array D with the marks "P" and "S" it is easy to see that each of these problems can be solved using the nearest-one algorithm of [4] in $O(\alpha(n))$ time with an optimal number of processors, and thus also in $O(\log^* n)$ time with an optimal number of processors. In fact it turns out that the prefix-minima and suffix-maxima information suffices for solving each of these three subproblems in $O(1)$ time using n processors (we omit the details). All other steps of the algorithm need $O(1)$ time and $O(n)$ operations.

Finally, we need to compute for each net its actual track number on the top and bottom of the rectangle. The track numbers for each net at the top can be obtained as follows. Observe that at the end of Step 3 of the algorithm, all top edges in D, match an element at i with an element at j where i is even and j is odd and $d_i > d_j$. This pair of elements is assigned track number $i/2$, with the element at location j having direction CC and the element at location i having direction C. Clearly this track assignment is unique for the matchings induced by each top edge. The track assignment at the bottom can be done in a similar manner by observing that at the end of Step 3 of the algorithm all bottom edges are of the form (d_{2j-1}, d_{2j}) for all $j, 1 \leq j \leq m/2$. □

3. The prefix-minima problem.

In this section we present a fast parallel algorithm for the prefix-minima problem: Given an array of integers $A = [a_1, a_2, \ldots, a_n]$ drawn from the range of integers $[1..s]$, find for each i, $1 \leq i \leq n$ the leftmost minimal element over the prefix $[a_1, a_2, \ldots, a_i]$.

We give two main results; both are optimal parallel algorithms. The first is a triply-logarithmic time algorithm for an arbitrary input range size s, and the second is an algorithm that runs in $O(\log^* n)$ time when s is equal to n.

We start by reducing the prefix-minima problem to the problem of finding critical elements, which is defined below. Later we give an algorithm for the problem of finding critical elements.

Finding critical elements. Given is an array of integers $A = [a_1, a_2, \ldots, a_n]$ drawn from the range of integers $[1..s]$. An element a_i, $1 \leq i \leq n$ is **critical** if $a_i < a_j$, for every j, $1 \leq j < i$. The problem is to find all critical elements in A.

The reduction. Given some element a_i, $1 \leq i \leq n$, let $c(i) \leq i$ be the maximal index such that $a_{c(i)}$ is critical. Note that a_{c_i} is the leftmost minimal element over the prefix $a_1, a_2, \ldots, a_i$. Thus, given all critical elements in A, the prefix-minima problem is equivalent to finding for each i, $1 \leq i \leq n$, the index $c(i)$. This computation can be done using the nearest-one algorithm of [4] in $O(\alpha(n))$ time (where $\alpha(n)$ is the inverse of Ackermann's function) using an optimal number of processors (and thus also in $O(\log^* n)$ time using an optimal number of processors).

3.1. Overview.

Our main tool is a basic algorithm for finding critical elements that evolves in two directions giving the two main results mentioned above.

This basic algorithm runs in time $O(1)$ using $n \log s$ processors and $O(s)$ space for an input of length n, drawn from the range of integers $[1..s]$.

Stages in the description of the triply-logarithmic algorithm.

(1) The basic algorithm.

(2) Reducing the space of the basic algorithm.

For any constant $0 < \epsilon < 1$, the resulting algorithm runs in time $O(1)$ using $\epsilon n \log s$ processors and $O(ns^\epsilon)$ space.

(3) The optimal triply-logarithmic algorithm.

The algorithm runs in $O(\log\log\log s)$ time using $n/\log\log\log s$ processors and $O(ns^\epsilon)$ space.

Stages in the description of the $O(\log^* n)$ time algorithm.

(1) The basic algorithm.

(2) An $O(\log^* s)$ time algorithm.

The algorithm uses $s + n\log^* n$ processors and $O(n+s)$ space for an input of length n, drawn from the range of integers $[1..s]$. It is obtained by a recursive application of the basic algorithm.

(3) The optimal $O(\log^* n)$ time algorithm.

The algorithm runs in $O(\log^* n)$ time using $n/\log^* n$ processors and $O(n)$ space for an input of length n, drawn from the range of integers $[1..n]$ (i.e., $s = n$).

3.2. The basic constant-time algorithm. Assume for simplicity that $s = 2^\alpha$ for some integer $\alpha > 0$.

THEOREM 3.1. *The algorithm for finding critical elements below runs in $O(1)$ time using $n\log s$ processors and $O(s)$ space.*

Step 1. Build a complete binary tree, T, whose leaves are the integer values $1, 2, \ldots, s$. Let 1 be the level of the leaves and $\log s + 1$ be the level of the root.

Let v be a node of T and let the interval of integers $[i, \ldots, j]$ be its leaves; denote by $l(v)$ the minimum index k, $1 \le k \le n$ in A such that $i \le a_k \le j$. Sometimes, we use the notation $l[i, \ldots, j]$ instead of $l(v)$. No ambiguity will arise.

Example. Let $A = [a_1, a_2, a_3, a_4] = [4, 3, 1, 8]$; $s = 8$ and T is given in Fig. 7. Now $l(a) = 3, l(b) = l(e) = l(g) = 1$, and $l(d) = l(f) = 4$ ($l(c)$ is undefined).

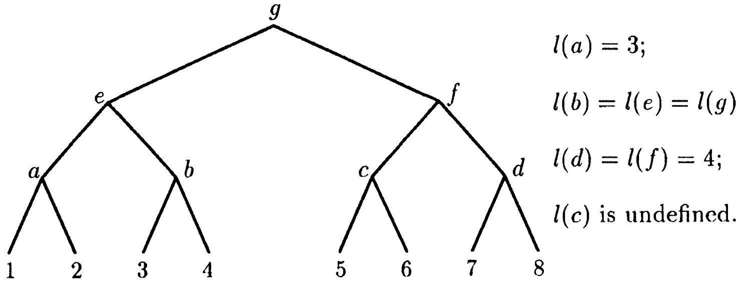


FIG. 7. *The tree T and its l-values.*

We allocate $\log s + 1$ processors to each element a_i in A, a processor for each ancestor of the *value* a_i in T (note that a node of T is defined to be one of its own ancestors).

Step 2. Compute $l(v)$ for each node v in T as follows. Each element a_i tries to write $l(v) := i$ for each ancestor v of the value a_i in T. (If two or more elements attempt to write at the same location, the element with minimal index in A succeeds.)

The computation in Step 3 below relies on the following two facts: (1) For every i, $1 \le i \le n$, a_i is critical if and only if there is no $j < i$ such that $a_j \le a_i$. To state the second fact we need the following definition. An interval of integers $[i, \ldots, j]$ is a *canonical interval* if there is a node of T such that the integer values $\{i, \ldots, j\}$ are all its leaves. (2) For every k, $1 \le k \le s$, the interval of integers $[1, \ldots, k]$ can be represented as a (disjoint) union of $\log s$ canonical intervals.

Step 3. For each element a_i test (in constant-time) whether it is critical, as follows. First, check whether $l(a_i) = i$. If not (implying that there exist an element a_j such that

$a_j = a_i$ and $j < i$) conclude that element a_i is not critical. If yes, consider the interval of integers $[1, 2, \ldots, a_i - 1]$ and take the $\leq \log s$ canonical intervals $[i_1, \ldots, j_1], [i_2, \ldots, j_2] \ldots [i_{\log s}, \ldots, j_{\log s}]$, whose union represents this interval. If $l[i_k, \ldots, j_k] < i$ for some k, $1 \leq k \leq \log s$, we conclude that a_i is not critical. Otherwise a_i is critical.

Assignment of the $\log s$ processors of a_i to the canonical intervals in Step 3 in constant time is standard, and is therefore omitted.

3.3. The optimal triply-logarithmic algorithm.

LEMMA 3.2. *There is an algorithm for finding critical elements that runs in $O(1)$ time using $n \log s$ processors and $O(ns^{\epsilon})$ space for any given constant ϵ, $0 < \epsilon < 1$.*

The algorithm that realizes the lemma above is based on adding a variant of the Radix sort idea, where the most significant bits are handled first, to the basic algorithm above. For details the reader is referred to the appendix.

THEOREM 3.3. *There is an algorithm for finding critical elements that runs in $O(\log\log\log s)$ time using an optimal number of processors and $O(ns^{\epsilon})$ space.*

The proof of the above theorem is easy. Partition the input array A into successive subarrays of $\log s$ elements each. Within each subarray find critical elements using a doubly-logarithmic optimal algorithm for prefix minima (see [3], for instance). This takes $O(\log\log\log s)$ time using $n/\log\log\log s$ processors. Each of these $n/\log s$ subarrays contributes its minimum to form an array B of elements. Informally, we finish by applying the algorithm of Lemma 3.2 to B to obtain the claimed bounds.

3.4. An optimal $O(\log^* n)$ time algorithm. In this subsection we consider the special case where the elements of the array A are restricted to the range of integers $[1..n]$. Unfortunately, we do not see a way to use the ideas introduced below to get a faster algorithm for a general domain s. The reason is that the number of processors we use is almost as large as the size of the domain (see, e.g., Lemma 3.5, where more than s processors are used).

THEOREM 3.4. *The algorithm below finds critical elements in $O(\log^* n)$ time using $n/\log^* n$ processors and linear space.*

In order to prove the above theorem we first prove the following lemma concerning a general input range $[1..s]$.

LEMMA 3.5. *There is an algorithm for finding critical elements that runs in $O(\log^* s)$ time using $s + n\log^* n$ processors and $O(n + s)$ space.*

Proof of Lemma 3.5. We begin with a high-level description followed by implementation details.

Step 1. For each k, $1 \leq k \leq s/\log s$, we select as a **representative** of the range of integers $[(k-1)\log s + 1..k \log s]$, the leftmost element a_i in A which belongs to the range (if such an element a_i exists). The number of selected representatives is at most $s/\log s$. We refer to a representative which is a critical element in A as a **critical representative**. To avoid confusion we note here that the computation below is done with respect to the original array A (and not with respect to any array of representatives).

Step 2. Find all critical representatives in A.

For i, $1 \leq i \leq n$, let left(i) denote the maximum index such that $a_{\text{left}(i)}$ is a critical representative and left$(i) < i$. Similarly, let right(i) denote the minimum index such that $a_{\text{right}(i)}$ is a critical representative and right$(i) > i$.

Step 3. For each element a_i, $1 \leq i \leq n$, find left(i) and right(i).

Consider some critical representative a_i in A. An element a_r, $i < r <$ {right(i), is critical if and only if a_r is critical in the subarray $a_i, a_{i+1}, \ldots, a_{\text{right}(i)-1}$ of A. Let $k = \lceil a_i/\log s \rceil$. In Step 4 below we finish the computation of critical elements in A by finding all critical elements in subarray $a_i, a_{i+1}, \ldots, a_{\text{right}(i)-1}$, for every critical representative a_i *separately*. The range of the elements for each of these separate subproblems is $[(k-1)\log s + 1..k \log s]$ (elements not in this range are not critical and can be given a dummy value of $k \log s$). For convenience we will view the range as $[1..\log s]$.

Step 4. Apply (recursively) Steps 1 to 4 to solve each of the separate subproblems.

Below we show how each of Steps 1 to 3 can be implemented in $O(1)$ time using $s + n \log^* n$ processors and $O(s+n)$ space. With respect to Step 4, we show how to allocate processors and space for each of the separate subproblems within the same time, processors and space bounds as Steps 1 to 3. After $\log^* s$ iterations of Steps 1 to 4 the size of the range for each of the separate subproblems is constant and the critical elements for each of these subproblems can be found in $O(1)$ time. Hence the whole algorithm takes $O(\log^* s)$ time, using $s + n \log^* n$ processors and $O(s+n)$ space. □

Implementation details.

Step 1. We use an auxiliary array B of size $s/\log s$. A processor that we allocate to each element a_i, $1 \le i \le n$, tries to write $B(\lceil a_i/\log s \rceil) := i$; The processor of a_i succeeds if and only if a_i is the leftmost element in A that belongs to the range of integers $[(\lceil a_i/\log s\rceil - 1)\log s + 1..\lceil a_i/\log s\rceil \log s]$.

Step 2. We apply the basic constant-time algorithm, but only with respect to the representatives (that is, among the elements of A we allocate processors to the representatives only). This takes $O(1)$ time using $s/\log s \cdot \log s = s$ processors and $O(s)$ space. Allocating $\log s$ processors to each representative as required by the basic constant-time algorithm is done using the array B obtained in Step 1 above.

Step 3. We use an auxiliary array $C = (c_1, c_2, \ldots, c_n)$ of bits that are all zeros initially. Each critical representative a_i, $1 \le i \le n$ writes $c_i := 1$. The problem then is to find the nearest ones (to the left and right) for each element in C. This can be done in $O(1)$ time using $n \log^* n$ processors using the nearest-one algorithm of [4].

Step 4. Consider the subproblem of finding all critical elements in the subarray $a_i, a_{i+1}, \ldots, a_{\mathrm{right}(i)-1}$ for some critical representative a_i in A. Allocating $(\mathrm{right}(i) - i)\log^* n$ processors (though only $(\mathrm{right}(i)-i)\log^*(\mathrm{right}(i)-i)$ are needed) and $O(\mathrm{right}(i)-i)$ space to this subproblem can be done using the output of Step 3. Allocating an additional $\log s$ processors and $O(\log s)$ space can be done using the array B that was computed in Step 1.

Proof of Theorem 3.4. We start with n elements, each assuming an integer value in the range $[1..n]$. The index of each element is an integer in the domain $[1\ldots n]$. Step 1 below reduces the size of the domain (of indices) from n to $n/(\log^* n)^2$. Step 2 reduces the size of the range (of values) from n to $n/(\log^* n)^2$. Step 3 applies the algorithm of Lemma 3.5 to find all critical elements in the reduced problem. Step 4 extends the solution of the reduced problem for finding all critical elements with respect to A.

Step 1. Partition the input array A into successive subarrays of $(\log^* n)^2$ numbers each. Find the minimum element in each subarray and put these minima in an array A' of size $n/(\log^* n)^2$. This step can be done in $O(\log^* n)$ time using $n/\log^* n$ processors.

Step 2. For each k, $1 \le k \le n/(\log^* n)^2$ we select as a representative of the range of integers $[(k-1)(\log^* n)^2+1..k(\log^* n)^2]$, the leftmost element a'_i in A' that belongs to the range. Get A'' from A' by replacing the value of the representative of $[(k-1)(\log^* n)^2+1..k(\log^* n)^2]$ by k; the value of nonrepresentatives will be set by default to $n/(\log^* n)^2$, which is large enough not to affect the computation.

Step 3. Find all critical elements in A''. By Lemma 3.5 this takes $O(\log^* x)$ time using $x + x\log^* x$ processors and $O(x+x)$ space, where $x = n/(\log^* n)^2$, or $O(\log^* n)$ time, using $n/\log^* n$ processors and at most $O(n)$ space.

Step 4. Mark in A every critical element of A''. Using the nearest-one algorithm of [4], find for each element a_i of A the nearest marked element to its left and the nearest marked element to its right (denoted $a_{\text{left}(i)}$ and $a_{\text{right}(i)}$, respectively, as before). This can be done in $O(\log^* n)$ time using an optimal number of processors.

We characterize the cases in which a_i is a critical element. Let the value of $a_{\text{left}(i)}$ be in the range of integers $[(k-1)(\log^* n)^2 + 1..k(\log^* n)^2]$ for some k.

Case 1. a_i and $a_{\text{right}(i)}$ do not belong to the same subarray (of size $(\log^* n)^2$). Observe that (a) a_i must be at least $(k-1)(\log^* n)^2 + 1$, and (b) a_i is a critical element if for every j, $left(i) \leq j < i, a_i < a_j$. (Checking whether condition (b) holds is called TEST 1, below.)

Case 2. a_i and $a_{\text{right}(i)}$ belong to the same subarray. Observe that (a) If $a_i \leq (k-1)(\log^* n)^2$, then it is a critical element (with respect to A) if it is a critical element with respect to its subarray. (Checking whether a_i is a critical element with respect to its own subarray is called TEST 2, below.) (b) If $a_i > (k-1)(\log^* n)^2$, then it is a critical element (with respect to A) if the following two conditions hold: (i) a_i is a critical element with respect to its subarray (can be checked by TEST 2), and (ii) for every j, left$(i) \leq j < i, a_i < a_j$ (TEST 1).

To complete the description of the algorithm, we show how to perform TEST 1 and TEST 2.

Step 5.1. Perform a prefix minimum computation with respect to each subarray of size $(\log^* n)^2$. This can be done in time $O(\log^* n)$ using $\log^* n$ processors. This provides for TEST 2 whenever needed.

Step 5.2. For every marked element, we will have a "bulletin board" $B[1 \ldots (\log^* n)^2]$ of size $(\log^* n)^2$. For every element a_i, we do the following. Let the value of $a_{\text{left}(i)}$ be in the range of integers $[(k-1)(\log^* n)^2 + 1..k(\log^* n)^2]$ for some k. If a_i is in this range, then the index i is written into $B(a_i - (k-1)(\log^* n)^2)$ using the convention that the smallest among the indices attempted to be written there is actually written (B is the bulletin board of $a_{\text{left}(i)}$). Then find critical elements with respect to each bulletin board. Finally, conclude that a_i passes TEST 1 if $B(a_i - (k-1)(\log^* n)^2) = i$ and is a critical element.

Overall Step 5 takes $O(\log^* n)$ time using $n/\log^* n$ processors and linear space. □

Appendix. Implementing Radix prefix minima. The appendix connects to its reference in §3.

High-level description. We use a variant of the Radix sort idea, where the most significant bits are handled first. The binary representation of each a_k, $1 \leq k \leq n$, needs $\log s$ bits. For each i, $0 \leq i \leq 1/\epsilon$ let a_k^i be the number represented by the $i\epsilon \log s$ most significant bits of a_k. Let A^i be the array $(a_1^i, \ldots, a_n^i)$. We find critical elements with respect to A in $1/\epsilon$ stages.

Guided by Radix sort, we compute in stage i, $1 \leq i \leq 1/\epsilon$, critical elements with respect to A^i. In addition we have a flag for each element that determines whether the element is still a **candidate** for becoming critical. An element a_j is a candidate after stage i if (1) it has not yet been identified to be critical after stage i, and (2) its leftmost $i\epsilon \log s$ bits are the same as those of the nearest critical element to its left (implying that later stages may still find a_j to be a critical element).

Base of the inductive construction. a_1 is the only critical element with respect to A^0. All other elements are candidates. (The reason is that a_k^0, $1 \leq k \leq n$ is an empty prefix of a_k.) This is the input to stage $i = 1$.

The inductive construction. We describe stage i, where the critical elements with respect to array A^{i-1} are "refined" into critical elements with respect to array A^i. Consider an interval $[\beta, \ldots, \gamma]$ of A such that a_β and a_γ are adjacent critical elements in A^{i-1} (that is, for every j, $\beta < j < \gamma$, a_j is not critical in A^{i-1}). Element a_j, $\beta < j < \gamma$, is critical in A^i if the following two conditions hold: (i) it is a candidate after stage $i-1$; and (ii) it is critical in the subarray $[a_\beta^i, \ldots, a_{\gamma-1}^i]$. Condition (i) can be verified from the output of stage $i-1$. To verify condition (ii) with respect to any j, $\beta < j < \gamma$, we find all critical elements in $[a^i_\beta, \dots, a^i_{\gamma-1}]$. For this we need only consider the ith $\epsilon \log s$ bits of the elements in $[a^i_\beta, \dots, a^i_{\gamma-1}]$. The reason is that the first $(i-1)\epsilon \log s$ bits are the same for all candidates (noncandidates can be given a default value of s^ϵ). This is done using the basic constant-time algorithm.

The output of the last stage (stage $1/\epsilon$) results in finding all critical elements with respect to A.

4. Implementation details. (1) In stage i we need to find for each element in A the nearest critical elements (with respect to A^{i-1}) to its left and right. This is done using the nearest-one algorithm of [4] in $O(1)$ time using $n \log^* n$ processors, which we can assume is at most $n \log s$ processors for the following reason. If $n \log^* n > n \log s$ then $s \leq \log n$. In such a case, the prefix-minima problem can be solved in constant time using n processors using a (single) "bulletin board" of values. This bulletin board is of size $\log n$ and the prefix-minima problem with respect to it can be solved in constant time using $\log^2 n$ processors.

(2) Allocation of processors and space to each subproblem with respect to an interval $[\beta, \dots, \gamma]$ is done as follows: We allocate $\log s$ processors and s^ϵ space to each entry in A. For the subproblem associated with interval $[\beta, \dots, \gamma]$, we use only the space allocated to entry β.

(3) Application of the basic constant-time algorithm to the ith $\epsilon \log s$ bits of $[a^i_\beta, \dots, a^i_{\gamma-1}]$ takes $O(1)$ time using $(\gamma - \beta + 1)\epsilon \log s$ processors and s^ϵ space. Overall, the running time is $O(1)$ using $\epsilon n \log s$ processors and ns^ϵ space.

REFERENCES

[1] B. Baker, *A provably good algorithm for the two module routing problem*, SIAM J. Comput.,· 15 (1986), pp. 162–188.

[2] P. Beame and J. Hastad, *Optimal bounds for decision problems on the CRCW PRAM*, in Proc. of the 19th Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1987, pp. 83–93.

[3] O. Berkman, B. Schieber, and U. Vishkin, *Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values*, J. Algorithms, 14 (1993), pp. 344–370.

[4] O. Berkman and U. Vishkin, *Recursive *-tree parallel data-structure*, in Proc. of the 30th IEEE Annual Symposium on Foundation of Computer Science, IEEE Press, Research Triangle Park, North Carolina, 1989, pp. 196–202.

[5] T. Gonzalez and S. Lee, *A linear time algorithm for optimal routing around a rectangle*, J. Assoc. Comput. Mach., 35 (1988), pp. 810–831.

[6] A. LaPaugh, *A polynomial time algorithm for optimal routing around a rectangle*, in Proc. of the 21st IEEE Annual Symposium on Foundation of Computer Science, IEEE Press, Syracuse, New York, 1980, pp. 282–293.

[7] Y. Matias and U. Vishkin, *Converting high probability into nearly-constant time – with applications to parallel hashing*, in Proc. of the 23rd Annual ACM Symposium on Theory of Computing, ACM Press, New Orleans, LA, 1991, pp. 307–316.

[8] F. Meyer auf def Heide and A. Wigderson, *The complexity of parallel sorting*, in Proc. of the 26th IEEE Annual Symposium on Foundation of Computer Science, IEEE Press, Portland, Oregon, 1985, pp. 532–540.

[9] P. Ragde, *Towards lower bounds for parallel computation over moderate sized domains*. manuscript, 1990.

[10] M. Sarrafzadeh and F. Preparata, *A bottom-up layout technique based on two-rectangle routing*, Integration, the VLSI Journal, 5 (1987), pp. 231–246.

[11] B. Schieber, *Design and analysis of some parallel algorithms*, Ph.D. thesis, Department of Computer Science, Tel Aviv University, 1987.

[12] L. Valiant, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

[13] U. Vishkin, *Structural parallel algorithmics*, in Proc. of the 18th International Colloquium on Automata, Languages and Programming, Springer-Verlag, Madrid, Spain, 1991, pp. 363–380.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 466–487, June 1994

FASTER APPROXIMATION ALGORITHMS FOR THE UNIT CAPACITY CONCURRENT FLOW PROBLEM WITH APPLICATIONS TO ROUTING AND FINDING SPARSE CUTS*

PHILIP KLEIN[†], SERGE PLOTKIN[‡], CLIFFORD STEIN[§] AND ÉVA TARDOS[¶]

Abstract. This paper describes new algorithms for approximately solving the concurrent multicommodity flow problem with uniform capacities. These algorithms are much faster than algorithms discovered previously. Besides being an important problem in its own right, the uniform-capacity concurrent flow problem has many interesting applications. Leighton and Rao used uniform-capacity concurrent flow to find an approximately "sparsest cut" in a graph and thereby approximately solve a wide variety of graph problems, including minimum feedback arc set, minimum cut linear arrangement, and minimum area layout. However, their method appeared to be impractical as it required solving a large linear program. This paper shows that their method might be practical by giving an $O(m^2 \log m)$ expected-time randomized algorithm for their concurrent flow problem on an m-edge graph. Raghavan and Thompson used uniform-capacity concurrent flow to solve approximately a channel width minimization problem in very large scale integration. An $O(k^{3/2}(m+n\log n))$ expected-time randomized algorithm and an $O(k \min\{n, k\}(m+n\log n)\log k)$ deterministic algorithm is given for this problem when the channel width is $\Omega(\log n)$, where k denotes the number of wires to be routed in an n-node, m-edge network.

Key words. multicommodity flow, approximation, concurrent flow, graph separators, VLSI routing

AMS subject classification. 68Q25, 90C08, 90C27

1. Introduction.

The multicommodity flow problem involves shipping several different commodities from their respective sources to their sinks in a single network with the total amount of flow going through an edge limited by its capacity. The amount of each commodity we wish to ship is called the *demand* for that commodity. An optimization version of this problem is the *concurrent flow* problem in which the goal is to find the maximum percentage z such that at least z percent of each demand can be shipped without violating the capacity constraints. Here we consider the concurrent flow problem with unit capacities. Observe that in this case, the problem is equivalent to the problem of finding a flow (disregarding capacities) that minimizes the maximum total flow on any edge (the *congestion*). Let m, n, and k be, respectively, the number of edges, nodes, and commodities for the input network.

In this paper, we give algorithms that, for any positive ϵ, find a solution whose congestion is no more than $(1+\epsilon)$ times the minimum congestion. Our algorithms significantly improve the time required for finding such approximately optimal solutions.

One contribution of this paper is the introduction of a randomization technique useful in iterative approximation algorithms. This technique enables each iteration to be carried out much more quickly than by using known deterministic methods.

*Received by the editors August 10, 1992; accepted for publication November 19, 1992.

[†] Computer Science Department, Brown University, Providence, Rhode Island 02912. The work of this author was supported in part by Office of Naval Research grant N00014-88-K-0243 and Defense Advanced Research Projects Agency grant N00039-88-C0113 while the author was at Harvard University, Cambridge, Massachusetts. Additional support provided by National Science Foundation Research Initiation Award CCR-901-2357 and by Office of Naval Research and Defense Advanced Research Projects Agency contract N00014-83-K-0146 and Defense Advanced Research Projects Agency order 6320, amendment 1.

[‡] Department of Computer Science, Stanford University, Stanford, California 94305. The work of this author was supported by National Science Foundation Research Initiation Award CCR-900-8226, United States Army Research Office grant DAAL-03-91-G-0102, and Office of Naval Research contract N00014–88–K–0166.

[§] Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The work of this author was supported by National Science Foundation PYI Award CCR-89-96272 with matching support from United Parcel Service, Sun, and AT&T Bell Laboratories Graduate Fellowship.

[¶] School of Operations Research, Cornell University, Ithaca, New York 14853. The work of this author was supported in part by a Packard Research Fellowship and the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS-8920550.

Part of our motivation in developing algorithms for concurrent flow derives from two important applications, finding sparsest cuts and finding a very large scale integration (VLSI) routing that minimizes channel width.

Leighton and Rao [14] showed how to use the solution of a unit-capacity concurrent flow problem to find an approximate "sparsest cut" of a graph. As a consequence, they and other researchers have developed polylog-times-optimal approximation algorithms for a wide variety of graph problems, including minimum area VLSI layout, minimum cut linear arrangement, minimum feedback arc set [14], optimal linear and higher-dimensional arrangement [7], minimum chordal fill [9], and single-processor scheduling [17].

The computational bottleneck of the method of Leighton and Rao is solving a unit-capacity concurrent flow problem with $O(n)$ commodities, each with unit demand. They appealed to linear programming techniques to show that the problem can be solved in polynomial time. The new approximation algorithm greatly improves the resulting running time.

THEOREM 1.1. *For any fixed $\epsilon > 0$, a $(1+\epsilon)$-factor approximation to the unit-capacity, unit-demand concurrent flow problem can be found by a randomized algorithm in $O((k+m)m\log m)$ expected time, where the constant depends on ϵ.*

As an application of this result we substantially reduce the time required for Leighton and Rao's method.

THEOREM 1.2. *An $O(\log n)$-factor approximation to the sparsest cut in a graph can be found by a randomized algorithm in expected $O(m^2 \log m)$ time.*

Previous to our work, the best algorithm for this problem and a running time of $O(n^{4.5}\sqrt{m}\log n)$ [21] and made use of linear programming techniques and fast matrix multiplication.

Another application of our approximation algorithm is to VLSI routing in graphs. Raghavan and Thompson [16] and Raghavan [15] considered the problem of routing two-terminal nets (essentially wires) in a graph so as to minimize approximately the *channel width*, i.e., the maximum number of nets routed through an edge. The computational bottleneck in their algorithms is solving a unit-capacity concurrent flow problem. Their algorithms require a better than constant ϵ approximation to the concurrent flow problem. In fact, the algorithm of Theorem 1.1 is a *fully polynomial approximation* algorithm, i.e., its running time depends polynomially on ϵ^{-1}.

THEOREM 1.3. *For any positive $\epsilon < 1$ that is at least inverse polynomial in n, a $(1+\epsilon)$-factor approximation to the unit-capacity concurrent flow problem can be found by a randomized algorithm in expected time $O((\epsilon^{-1}k + \epsilon^{-3}m)(m\log n + n\log^2 m))$ and by a deterministic algorithm in time $O((k+\epsilon^{-2}m)k(m\log m + n\log^2 m))$.*

An application of the algorithm of Theorem 1.3 is a significant improvement in the time needed to solve Raghavan and Thompson's problem.

THEOREM 1.4. *If $w_{\min}$ denotes the minimum achievable channel width and $w_{\min} = \Omega(\log m)$, a routing of width $w_{\min} + O(\sqrt{w_{\min}\log n})$ can be found by a randomized algorithm in expected time $O(k^{3/2}(m + n\log n))$ and by a deterministic algorithm in time $O(k\min\{n,k\}(m+n\log n)\log k)$.*

Our algorithms compare favorably to previous work. The concurrent flow problem can be formulated as a linear program in $O(mk)$ variables and $O(m+nk)$ constraints (see, for example, [18]). Linear programming can be used to solve the problem optimally in polynomial time. Kapoor and Vaidya [8] gave a method to speed up the matrix inversions involved in Karmarkar type algorithms for multicommodity flow problems; combining their technique with Vaidya's new linear programming algorithm using fast matrix multiplication [21] yields a time bound of $O(k^{3.5}n^3\sqrt{m}\log(nD))$ for the unit-capacity concurrent flow problem with integer demands (where D denotes the sum of the demands) and an $O(\sqrt{m}k^{2.5}n^2 \log(n\epsilon^{-1}D))$ bound for the approximation problem.

Shahrokhi and Matula [18] gave a combinatorial fully polynomial approximation scheme for the unit-capacity concurrent flow problem (which they called the concurrent flow problem with uniform capacities). Their algorithm runs in $O(nm^7\epsilon^{-5})$ time.

Our approach to solving concurrent flow problems is a modification of the framework originated by Shahrokhi and Matula [18]. The idea is to use a length function on the edges to reflect congestion and iteratively reroute flow from long (more congested) paths to short (less congested) paths. Our approach differs from that of Shahrokhi and Matula in several ways. We develop a framework of *relaxed optimality conditions* that allows us to measure the congestion on both a local and a global level, thereby giving us more freedom in choosing which flow paths to reroute at each iteration. We exploit this freedom by using a faster *randomized* method for choosing flow paths. In addition, this framework also allows us to achieve greater improvement as a result of each rerouting. In Table 1, we give upper bounds on the running times for our algorithms. Our actual bounds are slightly better than those in the table and are given in more detail in the remainder of the paper. Note that by use of various combinations of our techniques, we can obtain slightly better bounds than those stated in Theorems 1.1 and 1.3.

TABLE 1

Upper bounds on the running times of our algorithms. The actual bounds are slightly better.

Algorithm type	Running Time
Randomized, fixed ϵ	$O(m(k+m)\log n)$
Deterministic, fixed ϵ	$O(mk(k+m)\log n)$
Randomized, $1 < \epsilon < \frac{1}{\text{poly}(n)}$	$O(\epsilon^{-3}m(k+m)\log^2 n)$
Deterministic, $1 < \epsilon < \frac{1}{\text{poly}(n)}$	$O(\epsilon^{-2}mk(k+m)\log^2 n)$

An earlier version of this paper has appeared in [11]. In the earlier version the case when both the capacities and the demands are uniform was considered separately from the more general case when only the capacities are assumed to be uniform. The earlier version presented a fast algorithm for the first case and a factor of $\epsilon^{-1}m$ slower one for the more general case. In this version we extend the algorithm for the uniform demand case to work for the more general case with at most a logarithmic slowdown.

In subsequent work building on that described here, Leighton et al. [13] gave a fast approximation algorithm for concurrent flow with arbitrary capacities. That algorithm is faster than ours when the number of commodities is less than $\sqrt{m}$. It makes use of the randomized technique introduced in this paper. Also, in subsequent work, Goldberg [4] and Grigoriadis and Khachiyan [6] showed that by a modification of the randomized technique, one can reduce the running time's dependence on ϵ for both our algorithm and that of Leighton et al.

2. Preliminaries and definitions. In this section we define the concurrent flow problem, introduce our notation, and give some basic facts regarding the problem. Concurrent flow is a variant of multicommodity flow, and we start by giving a formal definition of the latter.

The multicommodity flow problem is the problem of shipping several different commodities from their respective sources to their sinks in a single network, while obeying capacity constraints. More precisely, an instance of the multicommodity flow problem consists of an undirected graph $G = (V, E)$, a nonnegative capacity $\text{cap}(vw)$ for every edge $vw \in E$, and a specification of k commodities, numbered 1 through k. The specification for commodity i consists of a source-sink pair $s_i, t_i \in V$ and a nonnegative integer demand $d(i)$. We will denote the maximum demand by $d_{\max}$, the total demand $\sum_i d(i)$ by D, the number of nodes by n, the number of edges by m, and the number of different sources by k^*. Notice that $k^* \leq n$. For notational convenience we assume that $m \geq n$ and that the graph G has no parallel edges. If there is an edge between nodes v and w, this edge is unique by assumption, and we denote it by vw. Note that vw and wv denote the same edge.

A *flow* f_i in G from node s_i to node t_i can be defined as a collection of paths from s_i to t_i, with associated real values. Let $\mathcal{P}_i$ denote a collection of paths from s_i to t_i in G, and let $f_i(P)$ be a nonnegative value for every P in $\mathcal{P}_i$. The *value* of the flow thus defined is $\sum_{P \in \mathcal{P}_i} f_i(P)$, which is the total flow delivered from s_i to t_i. The amount of flow through an edge vw is

$$f_i(vw) = \sum \{f_i(P) \ : \ P \in \mathcal{P}_i \text{ and } vw \in P\}.$$

A *feasible multicommodity flow* f in G consists of a flow f_i from s_i to t_i of value $d(i)$ for each commodity $1 \leq i \leq k$. We require that $f(vw) \leq \text{cap}(vw)$ for every edge $vw \in E$, where we use $f(vw) = \sum_{i=1}^{k} f_i(vw)$ to denote the total amount of flow on the edge vw.

We consider the optimization version of the multicommodity flow problem, called the *concurrent flow problem* and first defined by Shahrokhi and Matula [18]. In this problem the objective is to compute the maximum possible value z such that there is a feasible multicommodity flow with demands $z \cdot d(i)$ for every $1 \leq i \leq k$. We call z the *throughput* of the multicommodity flow. An equivalent formulation of the concurrent flow problem is to compute the maximum z such that there is a feasible flow with demands $d(i)$ and capacities $\text{cap}(vw)/z$.

In this paper we shall focus exclusively on the special case of *unit capacities*, in which all edge-capacities are equal. The problem of finding a maximum throughput z can be reformulated in this special case as follows: ignore capacities, and find a multicommodity flow f that satisfies the demands and minimizes $|f| = \max_{vw \in E} \{f(vw)\}$, the maximum total flow on any edge.

A multicommodity flow f satisfying the demands $d(i)$ is *ϵ-optimal* if $|f|$ is at most a factor $(1+\epsilon)$ more than the minimum possible $|f|$. The *approximation problem* associated with the unit-capacity concurrent flow problem is to find an ϵ-optimal multicommodity flow f. We shall assume implicitly throughout that ϵ is at least inverse polynomial in n and at most $1/10$. These assumptions are not very restrictive as they cover practically every case of interest. To find an ϵ-optimal flow where $\epsilon \geq 1/10$, one can just find a $1/10$-optimal flow. To find an ϵ-optimal flow when $1/\epsilon$ is greater than any polynomial in n, one can run our algorithm. It will work for arbitrarily small ϵ, however, the running time will be slower than the time bounds given, as we will need to manipulate numbers whose size is exponential in the input. However, if this amount of accuracy is desired, it is more sensible and efficient to use any polynomial time linear programming algorithm to solve the problem exactly.

One can define the analogous problem for directed graphs. Our algorithms, and the corresponding time bounds, easily extend to the directed case by replacing (undirected) edges by (directed) arcs and paths by directed paths. Henceforth, we will concentrate only on the undirected case.

Linear programming duality gives a characterization of the optimum solution to the concurrent flow problem. Let $\ell : E \rightarrow \mathbf{R}$ be a nonnegative *length* function. For nodes $v, w \in V$ let $\text{dist}_\ell(v, w)$ denote the length of the shortest path from v to w in G with respect to the length function ℓ. For a path P we shall use $\ell(P)$ to denote the length of P. We shall use $|\ell|_1$ to denote $\sum_{vw \in E} \ell(vw)$, the sum of the length of the edges. The following theorem is a special case of the linear programming duality theorem (see, for example, [18]).

THEOREM 2.1. *For a multicommodity flow f satisfying the demands $d(i)$ and a length function ℓ,*

$$
(1)\qquad
\begin{aligned}
|f||\ell|_1 \geq \sum_{vw\in E} f(vw)\ell(vw) &= \sum_{i=1}^{k}\sum_{vw\in E}\ell(vw)f_i(vw)\\
&= \sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P) \geq \sum_{i=1}^{k}\operatorname{dist}_\ell(s_i,t_i)d(i).
\end{aligned}
$$

Furthermore, a multicommodity flow f minimizes $|f|$ if and only if there exists a nonzero length function ℓ for which all of the above terms are equal.

The optimality (complementary slackness) conditions given by linear programming can be reformulated in terms of conditions on edges and paths.

THEOREM 2.2. *A multicommodity flow f has minimum $|f|$ if and only if there exists a nonzero length function ℓ such that*

(1) *for every edge $vw \in E$ either $\ell(vw) = 0$ or $f(vw) = |f|$, and*

(2) *for every commodity i and every path $P \in \mathcal{P}_i$ with $f_i(P) > 0$ we have $\ell(P) = \operatorname{dist}_\ell(s_i, t_i)$.*

The goal of our algorithms is to solve the approximation problem, i.e., to find a multicommodity flow f and a length function ℓ such that the largest term, $|f||\ell|_1$, in (1) is within a $(1+\epsilon)$ factor of the smallest term, $\sum_i \operatorname{dist}_\ell(s_i,t_i)d(i)$. In this case, we say that f and ℓ are ϵ-optimal. Note that if f and ℓ are ϵ-optimal, then clearly f is ϵ-optimal. In fact, a multicommodity flow f is ϵ-optimal if and only if there exists a length function ℓ such that f and ℓ are ϵ-optimal.

3. Relaxed optimality conditions. Theorems 2.1 and 2.2 give two (apparently) different characterizations of exact optimality. Our goal is to find a flow that satisfies a relaxed version of Theorem 2.1. In order to do so, we will introduce a relaxed version of Theorem 2.2, the complementary slackness conditions of linear programming. We will then show that these *relaxed optimality conditions* are sufficient to show that the first and last terms in (1) are within a $(1+\epsilon)$ factor, and hence the flow f is ϵ-optimal. Our notion of relaxed optimality is analogous to the notion of ϵ-optimality used by Goldberg and Tarjan in the context of the minimum-cost flow problem [5].

Let $\epsilon > 0$ be an error parameter, f a multicommodity flow, and ℓ a length function. Throughout this section we shall use ϵ' to denote $\frac{\epsilon}{7}$. We say that a path $P \in \mathcal{P}_i$ for a commodity i is *ϵ-good* if

$$\ell(P) - \operatorname{dist}_\ell(s_i,t_i) \leq \epsilon'\ell(P) + \epsilon'\frac{|f|}{\min\{D, kd(i)\}}|\ell|_1$$

and *ϵ-bad* otherwise. The intuition is that a flow path is ϵ-good if it is short in either a relative or an absolute sense, i.e., either it is almost as short as the shortest possible (s_i,t_i)-path or it is at most a small fraction of $|\ell|_1$. We use this notion in defining the following *relaxed optimality conditions* (with respect to a flow f, a length function ℓ, and an error parameter ϵ):

(R1) For every edge $vw \in E$ either $\ell(vw) \leq \frac{\epsilon'}{m}|\ell|_1$ or $f(vw) \geq |f|/(1+\epsilon')$.

$$
(\text{R2})\qquad \sum_{i=1}^{k}\sum_{\substack{P\in\mathcal{P}_i\\ P\ \epsilon\text{-bad}}} f_i(P)\ell(P) \leq \epsilon'\sum_{i=1}^{k}\sum_{P\in\mathcal{P}_i} f_i(P)\ell(P).
$$

The first condition says that every edge either has a length that is a small fraction of the sum of the lengths of all edges or is almost saturated. The second condition says that the amount of flow that is on ϵ-bad paths, i.e., long paths, contributes a small fraction of the sum $f\cdot\ell$.

The next two lemmas show that the relaxed optimality conditions are sufficient to imply ϵ-optimality. We will first show that Condition (R1) implies that the first two terms in (1) are close. Then we will show that the two conditions together imply that the first and last terms in (1) are close. Thus we can conclude that the relaxed optimality conditions are sufficient to imply ϵ-optimality.

LEMMA 3.1. *Suppose a multicommodity flow f and a length function ℓ satisfy relaxed optimality condition* (R1). *Then*

$$(1-\epsilon')|f||\ell|_1 \le (1+\epsilon')\sum_{vw} f(vw)\ell(vw). \tag{2}$$

Proof. We estimate $|f||\ell|_1 = \sum_{vw}|f|\ell(vw)$ in two parts. The first part is the sum of the terms contributed by edges that satisfy $|f| \le (1+\epsilon')f(vw)$. This part of the sum is clearly at most $(1+\epsilon')\sum_{vw} f(vw)\ell(vw)$. If vw is an edge whose contribution is not counted in the first part, then, by assumption, $\ell(vw) \le (\epsilon'/m)|\ell|_1$. Therefore, the sum of all other terms is at most $\epsilon'|f||\ell|_1$. Thus, $|f||\ell|_1 \le (1+\epsilon')\sum_{vw} f(vw)\ell(vw) + \epsilon'|f||\ell|_1$. This implies the lemma. □

THEOREM 3.2. *Suppose f and ℓ and ϵ satisfy the Relaxed Optimality Conditions* (R1) *and* (R2). *Then f is ϵ-optimal, i.e., $|f|$ is at most a factor* $(1+\epsilon)$ *more than the minimum possible.*

Proof. We need to estimate the ratio of the terms in inequality (1) of Theorem 2.1. Lemma 3.1 estimates the ratio of the first two terms. We shall use this in estimating the ratio of the first and the last terms.

Consider the penultimate term in (1). We break this sum, $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P)$, into two parts: the sum over ϵ-good paths and the sum over ϵ-bad paths. Relaxed optimality condition (R2) gives us an upper bound of $\epsilon'\sum_i\sum_{P\in\mathcal{P}_i}\ell(P)f_i(P)$ on the sum over the ϵ-bad paths, and the definition of an ϵ-good path gives us the following bound on the sum over the ϵ-good paths:

$$\sum_{i=1}^{k} \sum_{\substack{P\in\mathcal{P}_i\\ P\ \epsilon\text{-good}}} f_i(P)\ell(P)$$
$$\le (1-\epsilon')^{-1}\sum_i\sum_{P\in\mathcal{P}_i}\left(\mathrm{dist}_\ell(s_i,t_i)f_i(P) + \epsilon'\frac{|f|}{\min\{D,kd(i)\}}|\ell|_1 f_i(P)\right).$$

Observing that $(\min\{D,kd(i)\})^{-1} \le D^{-1} + (kd(i))^{-1}$ and $\sum_{P\in\mathcal{P}_i} f_i(P) = d(i)$, we can bound the sum over the ϵ-good paths by

$$\sum_{i=1}^{k} \sum_{\substack{P\in\mathcal{P}_i\\ P\ \epsilon\text{-good}}} f_i(P)\ell(P)$$
$$\le (1-\epsilon')^{-1}\left(\sum_i \mathrm{dist}_\ell(s_i,t_i)d(i) + \sum_i \epsilon'|f||\ell|_1 d(i)\left(\frac{1}{D}+\frac{1}{kd(i)}\right)\right)$$
$$= (1-\epsilon')^{-1}\left(\sum_i \mathrm{dist}_\ell(s_i,t_i)d(i) + \sum_i \epsilon'|f||\ell|_1\left(\frac{d(i)}{D}+\frac{1}{k}\right)\right).$$

Now observe that there are exactly k commodities and $\sum_i d(i) = D$, so the last term sums to exactly $2\epsilon'|f||\ell|_1$. This gives that

$$\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \leq (1-\epsilon')^{-1}\left(\sum_i \text{dist}_\ell(s_i, t_i)d(i) + 2\epsilon'|f||\ell|_1\right).$$

Combining the bounds on the sum over ϵ-bad and ϵ-good paths we get

$$\begin{aligned}\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P) &\leq (1-\epsilon')^{-1}\sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_i \\ P\ \epsilon\text{-good}}} f_i(P)\ell(P) \\ &\leq (1-\epsilon')^{-2}\sum_i \text{dist}_\ell(s_i, t_i)d(i) + 2(1-\epsilon')^{-2}\epsilon'|f||\ell|_1.\end{aligned}$$

By the middle equations in Theorem 2.1, $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P)$ is equal to $\sum_{vw\in E} f(vw)\ell(vw)$. Lemma 3.1 gives a bound on $\sum_{vw\in E} f(vw)\ell(vw)$ in terms on $|f||\ell|_1$. Combining these inequalities and rearranging terms we get

$$\left(\frac{(1-\epsilon')^2}{1+\epsilon'} - \frac{2\epsilon'}{1-\epsilon'}\right)|f||\ell|_1 \leq \frac{1}{1-\epsilon'}\sum_i \text{dist}_\ell(s_i, t_i)d(i).$$

Combining the fractions and dropping low-order terms we get that

$$|f||\ell|_1 \leq \frac{1+\epsilon'}{1-5\epsilon'}\sum_i \text{dist}_\ell(s_i, t_i)d(i).$$

The assumption that $\epsilon \leq 1/10$ implies that $\epsilon' \leq 1/70$, which in turn implies that the factor $(1+\epsilon')/(1-5\epsilon')$ is less than $(1+7\epsilon') = (1+\epsilon)$. We combine this bound with inequality (1) to complete the proof. □

In the next two sections, we will focus on algorithms that achieve the relaxed optimality conditions.

4. Generic rerouting. In this section, we describe the procedure REDUCE that is the core of our approximation algorithms and prove bounds on its running time. Given a multicommodity flow f, procedure REDUCE modifies f until either f becomes ϵ-optimal or $|f|$ is reduced below a given target value. The approximation algorithms presented in the next two sections repeatedly call procedure REDUCE to decrease $|f|$ by a factor of 2, until an ϵ-optimal solution is found.

The basic step in our algorithms is choosing a flow path and rerouting some flow from this path to a "better" path. This step closely resembles the basic step in the algorithm of Shahrokhi and Matula [18]. The main differences are in the way we choose the paths and in the amount of flow that is rerouted at each iteration.

The key idea is to measure how good the current flow is by using the notion of ϵ-optimality, described in the previous section. Given a flow f and a value α to be determined later, we use a length function defined by $\ell(vw) = e^{\alpha f(vw)}$, which reflects the congestion of the edge vw. In other words, the length of an edge depends on the flow carried by the edge. Given an input ϵ, our algorithms gradually update f until f and ℓ (defined by the above formula) become ϵ-optimal. Each update is done by choosing an ϵ-bad flow path, rerouting some flow from this path to a much shorter path (with respect to ℓ), and recomputing the length function. We will prove below that the parameter α in the definition of length can be selected so that relaxed optimality condition (R1) is always satisfied. Through iterative reroutings of flow, we gradually enforce relaxed optimality condition (R2). When both relaxed optimality conditions are satisfied then Theorem 3.2 can be used to infer that f is ϵ-optimal.

For simplicity of presentation, we shall assume for now that the value of the length function $\ell(vw) = e^{\alpha f(vw)}$ at an edge vw can be computed in one step from $f(vw)$ and represented in a single computer word. In §4.3 we will remove this assumption and show that it is sufficient to compute an approximation to this value and show that the time required for computing a sufficiently good approximation does not change the asymptotic running times of our algorithms.

Procedure REDUCE (see Fig. 1) takes as input a multicommodity flow f, a target value τ, an error parameter ϵ, and a flow quantum σ_i for each commodity i. We require that each flow path comprising f_i carries flow that is an integer multiple of σ_i. The procedure repeatedly reroutes σ_i units of flow from an ϵ-bad path of commodity i to a shortest path. We will need a technical *granularity condition* that σ_i is small enough for every i to guarantee that approximate optimality is achievable through such reroutings. In particular, we assume that upon invocation of REDUCE, for every commodity i we have

$$\sigma_i \le \epsilon^2 \frac{\tau}{102 \log(7m\epsilon^{-1})}. \tag{3}$$

Upon termination, the procedure outputs an improved multicommodity flow f such that either $|f|$ is less than the target value τ or f is ϵ-optimal. (Recall that we have assumed that $\epsilon \le 1/10$.)

In the remainder of this section, we analyze the procedure REDUCE shown in Fig. 1. First, we show that throughout REDUCE f and ℓ satisfy relaxed optimality condition (R1). Second, we show that if the granularity condition is satisfied, the number of iterations in REDUCE is small. Third, we give an even smaller bound on the number of iterations for the case in which the flow f is $O(\epsilon)$-optimal upon invocation of REDUCE. This bound will be used in §5 to analyze an ϵ-scaling algorithm presented there. Fourth, we describe efficient implementations of procedure FINDPATH.

```
REDUCE($f$, $\tau$, $\epsilon$, $\sigma_i$ for $i = 1, \dots, k$)
$\alpha \leftarrow (7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$.
While $|f| \ge \tau$ and $f$ and $\ell$ are not $\epsilon$-optimal,
  For each edge $vw$, $\ell(vw) \leftarrow e^{\alpha f(vw)}$.
  Call FINDPATH($f$, $\ell$, $\epsilon$) to find an $\epsilon$-bad flow path $P$ and a short path $Q$
    with the same endpoints as $P$.
  Reroute $\sigma_i$ units of flow from $P$ to $Q$.
Return $f$.
```

FIG. 1. *Procedure* REDUCE.

4.1 Bounding the number of iterations of REDUCE.

LEMMA 4.1. *If f is a multicommodity flow and $\alpha \ge (7+\epsilon)|f|^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$, then the multicommodity flow f and the length function $\ell(vw) = e^{\alpha f(vw)}$ satisfy relaxed optimality condition* (R1).

Proof. Assume $|f| - f(v,w) \ge \frac{\epsilon}{7} f(v,w)$ for an edge $vw \in E$, and let ϵ' denote $\frac{\epsilon}{7}$. Observe that $|\ell|_1 \ge e^{\alpha|f|}$. Hence, we have

$$\frac{|\ell|_1}{\ell(v,w)} \geq \frac{e^{\alpha|f|}}{e^{\alpha f(v,w)}} \geq \frac{e^{\alpha|f|}}{e^{\alpha|f|(1+\epsilon')^{-1}}} = e^{\alpha|f|\epsilon'(1+\epsilon')^{-1}}.$$

We can use the bound on α in the statement of the lemma to conclude that this last term is at least $\frac{7m}{\epsilon}$. Thus, $\ell(vw) \leq \frac{\epsilon}{7m}|\ell|_1$. □

At the beginning of REDUCE, α is set equal to $(7+\epsilon)\tau^{-1}\epsilon^{-1}\log(7m\epsilon^{-1})$. As long as $|f| \geq \tau$, the value of α is sufficiently large, so by Lemma 4.1, relaxed optimality condition (R1) is satisfied. If we are lucky and relaxed optimality condition (R2) is also satisfied, then it follows that f and ℓ are ϵ-optimal. Now we show that if (R2) is not satisfied, then we can make significant progress. Like Shahrokhi and Matula, we use $|\ell|_1$ as a measure of progress.

LEMMA 4.2. *Suppose σ_i and τ satisfy the granularity condition. Then rerouting σ_i units of flow from an ϵ-bad path of commodity i to the shortest path with the same endpoints decreases $|\ell|_1$ by*

$$\Omega\left(\frac{\sigma_i}{\min\{D, kd(i)\}}|\ell|_1 \log m\right).$$

Proof. Let P be an ϵ-bad path from s_i to t_i, and let Q be a shortest (s_i,t_i)-path. Let $A = P - Q$ and $B = Q - P$. The only edges whose length changes due to the rerouting are those in $A \cup B$. The decrease in $|\ell|_1$ is $\ell(A) + \ell(B) - e^{-\alpha\sigma_i}\ell(A) - e^{\alpha\sigma_i}\ell(B)$, which can also be written as

$$(1-e^{-\alpha\sigma_i})(\ell(A)-\ell(B)) - (1-e^{-\alpha\sigma_i})(e^{\alpha\sigma_i}-1)\ell(B).$$

The granularity condition, the definition of α, and the assumption that $\epsilon \leq 1/10$ imply that $\alpha\sigma_i \leq \frac{7+\epsilon}{102}\epsilon \leq \epsilon/7 \leq 1/70$. For $0 \leq x \leq \frac{1}{70}$, we have $e^x \geq 1+x$, $e^x \leq 1+\frac{141}{140}x$, and $e^{-x} \leq 1-\frac{139}{140}x$. Thus the decrease is at least

$$\frac{139}{140}\alpha\sigma_i\,(\ell(A)-\ell(B)) - (\alpha\sigma_i)\left(\frac{141}{140}\alpha\sigma_i\right)\ell(B).$$

Now, observe that $\ell(A)-\ell(B)$ is the same as $\ell(P)-\ell(Q)$ and that $\ell(Q) = \mathrm{dist}_\ell(s_i,t_i)$. Also $\ell(B) \leq \ell(P)$. This gives a lower bound of

$$\frac{139}{140}\alpha\sigma_i\,(\ell(P)-\mathrm{dist}_\ell(s_i,t_i)) - \left(\frac{141}{140}\alpha^2\sigma_i^2\right)\ell(P).$$

But P is ϵ-bad, so this must be at least

$$\frac{139}{140}\alpha\sigma_i(\epsilon'\ell(P) + \epsilon'\frac{|f|}{\min\{D,kd(i)\}}|\ell|_1) - \frac{141}{140}\alpha^2\sigma_i^2\ell(P)$$

$$= \frac{139}{140}\alpha\sigma_i\epsilon'\ell(P) - \frac{141}{140}\alpha^2\sigma_i^2\ell(P) + \frac{139}{140}\alpha\epsilon'|f|\frac{\sigma_i}{\min\{D,kd(i)\}}|\ell|_1.$$

We have seen that $\frac{7+\epsilon}{102}\epsilon \geq \alpha\sigma_i$, which implies that $139\epsilon' \geq 141\alpha\sigma_i$, and therefore the first term dominates the second term. Thus the third term gives a lower bound on the decrease in $|\ell|_1$.

Substituting the value of α and using the fact that during execution of REDUCE we have $\tau \leq |f|$ yields the claim of the lemma. □

The following theorem bounds the number of iterations in REDUCE.

THEOREM 4.3. *If, for every commodity i, τ and σ_i satisfy the granularity condition and $|f| = O(\tau)$ initially, then the procedure* REDUCE *terminates after*

$$O(\epsilon^{-1} \max_i \frac{\min\{D, kd(i)\}}{\sigma_i})$$

iterations.

Proof. Theorem 3.2 implies that if f and ℓ satisfy the relaxed optimality conditions, then they are ϵ-optimal. By Lemma 4.1, relaxed optimality condition (R1) is maintained throughout all iterations. The fact that f is not yet ϵ-optimal implies that condition (R2) is not yet satisfied. Hence there exists an ϵ-bad path for FINDPATH to find. A single rerouting of flow from an ϵ-bad path of commodity i to a shortest path results in a reduction in $|\ell|_1$ of at least

$$\Omega\left(\frac{\sigma_i}{\min\{D, kd(i)\}}|\ell|_1 \log(m\epsilon^{-1})\right).$$

Since $1 - x < e^{-x}$, it follows that every

$$O\left(\max_i \frac{\min\{D, kd(i)\}}{\sigma_i} \log^{-1}(m\epsilon^{-1})\right)$$

iterations reduce $|\ell|_1$ by at least a constant factor.

Next we bound the number of times $|\ell|_1$ can be reduced by a constant factor. Let f' denote the input multicommodity flow. For every edge vw, $f'(vw) \leq |f'|$. Hence after we first assign lengths to edges, the value of $|\ell|_1$ is at most $me^{\alpha|f'|}$. The length of every edge remains at least 1, so $|\ell|_1$ is always at least m. Therefore, $|\ell|_1$ can be reduced by a factor of e at most $\alpha|f'|$ times, which is $O(\epsilon^{-1} \log(m\epsilon^{-1}))$ by the assumption that $f = O(\tau)$ and the value of α. This proves that REDUCE terminated in the claimed number of iterations. □

THEOREM 4.4. *Suppose that the input flow f is $O(\epsilon)$-optimal, σ and τ satisfy the granularity condition, and $|f| = O(\tau)$ initially. Then the procedure* REDUCE *terminates after*

$$O\left(\max_i \frac{\min\{D, kd(i)\}}{\sigma_i}\right)$$

iterations.

Proof. Again let f' denote the input multicommodity flow. The assumption that f' is $O(\epsilon)$-optimal implies that $|f'| \leq (1 + O(\epsilon))|f|$ for *every* multicommodity flow f. Therefore, the value of $|\ell|_1$ is never less than $e^{(1+O(\epsilon))^{-1}\alpha|f'|}$. As in Theorem 4.3, the initial value of $|\ell|_1$ is at most $me^{\alpha|f'|}$, so the number of times $|\ell|_1$ can be reduced by a constant factor is $O(\alpha\epsilon|f'| + \log m)$, which is $O(\alpha\epsilon|f'|)$ by the choice of α and τ. The theorem then follows as in the proof of Theorem 4.3. □

4.2. Implementing an iteration of REDUCE. We have shown that REDUCE terminates after a small number of iterations. It remains to show that each iteration can be carried out quickly. REDUCE consists of three steps—computing lengths, executing FINDPATH, and rerouting flow. We discuss computing lengths in §4.3. In this section, we discuss the other two steps.

We now consider the time taken by procedure FINDPATH. We will give three implementations of this procedure. First, we will give a simple deterministic implementation that runs in $O(k^*(m + n \log n) + n \sum_i (d(i)/\sigma_i))$ time, then a more sophisticated implementation that runs in time $O(k^*n\log n + m(\log n + \min\{k, k^*\log d_{\max}\}))$, and finally a randomized implementation that runs in expected $O(\epsilon^{-1}(m + n\log n))$ time. All of these algorithms use the shortest-paths algorithm of Fredman and Tarjan [3] that runs in $O(m + n\log n)$ time.

To find a bad flow path deterministically, we first compute, for every source node s_i, the length of the shortest path from s_i to every other node v. This takes $O(k^*(m + n\log n))$ time. In the simplest implementation we then compute the length of every flow path in $\mathcal{P}$ and compare its length to the length of the shortest path to decide if the path is ϵ-bad. There could be as many as $\sum_i (d(i)/\sigma_i)$ flow paths, each consisting of up to n edges; hence, computing these lengths takes $O(n\sum_i(d(i)/\sigma_i))$ time.

To decrease the time required for FINDPATH we have to find an ϵ-bad path, if one exists, without computing the length of so many paths. Observe that if there is an ϵ-bad flow path for commodity i, then the longest flow path for commodity i must be ϵ-bad. Thus, instead of looking for an ϵ-bad path in $\mathcal{P}_i$ for some commodity i, it suffices to find an ϵ-bad path in the directed graph obtained by taking all flow paths in $\mathcal{P}_i$ and treating the paths as directed away from s_i. In order to see if there is an ϵ-bad path we need to compute the length of the longest path from s_i to t_i in this directed graph. To facilitate this computation we shall maintain that the directed flow graph is acyclic.

Let G_i denote the flow graph of commodity i. If G_i is acyclic, an $O(m)$ time dynamic programming computation suffices to compute the longest paths from s_i to every other node. Suppose that in an iteration we reroute flow from an ϵ-bad path from s_i to t_i, in the flow graph G_i. We must first update the flow graph G_i to reflect this change. Second, the update might introduce directed cycles in G_i, so we must eliminate such cycles of flow. We use an algorithm due to Sleator and Tarjan [19] to implement this process. Sleator and Tarjan gave a simple $O(nm)$ algorithm and a more sophisticated $O(m\log n)$ algorithm for the problem of converting an arbitrary flow into an acyclic flow.

Note that eliminating cycles only decreases the flows on edges, so it cannot increase $|\ell|_1$. Thus our bound on the number of iterations in REDUCE still holds.

We compute the total time required for each iteration of REDUCE as follows. In order to implement FINDPATH, we must compute the shortest path from s_i to t_i in G and the longest path from s_i to t_i in G_i for every commodity i, so the time required is $O(k^*(m+n\log n)+km)$. Furthermore, after each rerouting, we must update the appropriate flow graph and eliminate cycles. Elimination of cycles takes $O(m\log n)$ time. Combining these bounds gives an $O(k^*n\log n + m(k + \log n))$ bound on the running time of FINDPATH.

In fact, further improvement is possible if we consider the flow graphs of all commodities with the same source and same flow quantum σ_i together. Let $G_{v,\sigma}$ be the directed graph obtained by taking the union of all flow paths $P \in \mathcal{P}_i$ for a commodity i with $s_i = v$ and $\sigma_i = \sigma$, treating each path as directed away from v. If $G_{v,\sigma}$ is acyclic, an $O(m)$ time dynamic programming computation suffices to compute the longest paths from v to every other node in $G_{v,\sigma}$.

During our concurrent flow algorithm all commodities with the same demand will have the same flow quantum. To limit the different flow graphs that we have to consider we want to limit the number of different demands. By decomposing demand $d(i)$ into at most $\log d(i)$ demands with source s_i and sink t_i we can assume that each demand is a power of 2. This way the number of different flow graphs that we have to maintain is at most $k^*\log d_{\max}$.

LEMMA 4.5. *The total time required for deterministically implementing an iteration of* REDUCE *(assuming that exponentiation is a single step) is* $O(k^*n\log n + m(\log n + \min\{k, k^*\log d_{\max}\}))$.

Next, we give a randomized implementation of FINDPATH that is much faster when ϵ is not too small; this implementation seems simple enough to be practical. If f and ℓ are not ϵ-

optimal, then relaxed optimality condition (R2) is not satisfied, and thus ϵ-bad paths contribute at least an $\frac{\epsilon}{7}$-fraction of the total sum $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P) f_i(P)$. Therefore, by randomly choosing a flow path P with probability proportional to its contribution to the above sum, we have at least an $\frac{\epsilon}{7}$ chance of selecting an ϵ-bad path. Furthermore, we will show that we can select a candidate ϵ-bad path according to the right probability in $O(m)$ time. Then we can compute a shortest path with the same endpoints in $O(m + n \log n)$ time. This enables us to determine whether or not P was an ϵ-bad path. Thus we can implement FINDPATH in $O(\epsilon^{-1}(m+n \log n))$ expected time.

The contribution of a flow path P to the above sum is just the length of P times the flow on P, so we must choose P with probability proportional to this value. In order to avoid examining all such flow paths explicitly, we use a two-step procedure, as described in the following lemma.

LEMMA 4.6. *If we choose an edge vw with probability proportional to $\ell(vw) f(vw)$ and then select a flow path among paths through this edge vw with probability proportional to the value of the flow carried on the path, then the probability that we have selected a given flow path P is proportional to its contribution to the sum $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P) f_i(P)$.*

Proof. Let $B = \sum_i \sum_{P\in\mathcal{P}_i} \ell(P) f_i(P)$. Select an edge vw with probability $f(vw)\ell(vw)/B$. Once an edge vw is selected, choose a path $P \in \mathcal{P}_i$ through edge vw with probability $\frac{f_i(P)}{f(vw)}$. Consider a commodity i and a path $P \in \mathcal{P}_i$.

$$Pr(P \text{ chosen}) = \sum_{vw\in P} Pr(vw \text{ chosen}) \times \frac{f_i(P)}{f(vw)} = \sum_{vw\in P} \frac{f(vw)\ell(vw)}{B} \times \frac{f_i(P)}{f(vw)}$$

$$= \sum_{vw\in P} \frac{\ell(wv) f_i(P)}{B} = \frac{f_i(P)\ell(P)}{B}. \qquad \square$$

Choosing an edge with probability proportional to $\ell(vw) f(vw)$ can easily be done in $O(m)$ time. In order to choose with the right probability a flow path going through that edge, we need a data structure to organize these flow paths. For each edge we maintain a balanced binary tree with one leaf for each flow path through the edge, labeled with the flow value of that flow path. Each internal node of the binary tree is labeled with the total flow value of its descendent leaves. The number of paths is polynomial in n and ϵ^{-1}; therefore, using this data structure, we can randomly choose a flow path through a given edge in $O(\log n)$ time.

In order to maintain this data structure, each time we change the flow on an edge, we must update the binary tree for that edge, at a cost of $O(\log n)$ time. In one iteration of REDUCE the flow only changes on $O(n)$ edges; therefore, the time to do these updates is $O(n \log n)$ per call to FINDPATH, which is dominated by the time to compute single-source shortest paths.

We have shown that if relaxed optimality condition (R2) is not satisfied, then, with probability of at least $\epsilon/7$, we can find an ϵ-bad path in $O(m + n \log n)$ time. FINDPATH continues to pick paths until either an ϵ-bad path is found or $7/\epsilon$ trials are made. Observe that given that f and ℓ are not yet ϵ-optimal (which implies that condition (R2) is not yet satisfied), the probability of failure to find an ϵ-bad path in $7/\epsilon$ trials is bounded by $1/e$. Thus, in this case, REDUCE can terminate, claiming that f and ℓ are ϵ-optimal with probability of at least $1 - 1/e$. Computing lengths and updating flows can each be done in $O(n \log n)$ time, thus we get the following bound:

LEMMA 4.7. *One iteration of* REDUCE *can be implemented randomly in time* $(\epsilon^{-1}(m + n \log n))$ *time* (*assuming that exponentiation is a single step*).

The randomized algorithm as it stands is *Monte Carlo*; there is a nonzero probability that REDUCE erroneously claims to terminate with an ϵ-optimal f. To make the algorithm *Las Vegas* (never wrong, sometimes slow), we introduce a deterministic check. If FINDPATH fails to find an ϵ-bad path, REDUCE computes the sum $\sum_i \text{dist}_\ell(s_i, t_i)d(i)$ to the required precision and compares it with $|f||\ell|_1$ to determine whether f and ℓ are really ϵ-optimal. If not, the loop resumes. The time required to compute the sum is $O(k^*(m + n\log n))$, because at most k^* single-source shortest path computations are required. The probability that the check must be done t times in a single call to REDUCE is at most $(e^{-1})^{t-1}$, so the total expected contribution to the running time of REDUCE is at most $O(k^*(m + n\log n))$.

Recall that the bound on the number of iterations of REDUCE is greater than

$$\max_i \frac{\min\{D, kd(i)\}}{\sigma_i},$$

which in turn is at least k. Since in each iteration we carry out at least one shortest path computation, the additional time spent on checking does not asymptotically increase our bound on the running time for REDUCE.

We conclude this section with a theorem summarizing the running time of REDUCE for some cases of particular interest. For all of these bounds the running time is computed by multiplying the appropriate time for an iteration of REDUCE by the appropriate number of iterations of REDUCE. These bounds depend on the assumption that exponentiation is a single step. In §4.3 we shall show that the same bounds can be achieved without this assumption. We shall also give a more efficient implementation for the case when ϵ is a constant.

THEOREM 4.8. *Let $f = O(\tau)$ and τ and σ_i satisfy the granularity condition. Let*

$$H(k, d, \sigma) = \max_i \frac{\min\{D, kd(i)\}}{\sigma_i},$$

and let $\hat{k} = \min\{k, k^* \log d_{\max}\}$. Then the following table contains running times for various implementations of procedure REDUCE *(assuming that exponentiation is a single step).*

	Randomized Implementation	Deterministic Implementation
$1 \le \epsilon \le \frac{1}{poly(n)}$	$O\left(\epsilon^{-2}(m + n\log n)H(k, d, \sigma)\right)$	$O\left(\epsilon^{-1}H(k, d, \sigma)\right.$ $\left.[k^*n\log n + m(\log n + \hat{k})]\right)$
$1 \le \epsilon \le \frac{1}{poly(n)}$, f is $O(\epsilon)$-opt.	$O\left(\epsilon^{-1}(m + n\log n)H(k, d, \sigma)\right)$	$O\left(H(k, d, \sigma)\right.$ $\left.[k^*n\log n + m(\log n + \hat{k})]\right)$

4.3. Further implementation details. In this section, we will show how to get rid of the assumption that exponentiation can be performed in a single step. We will also give a more efficient implementation of the procedure REDUCE for the case when ϵ is fixed.

4.3.1. Removing the assumption that exponentiation can be performed in $O(1)$ time. To remove the assumption that exponentiation can be performed in $O(1)$ time, we will need to do two things. First we will show that it is sufficient to work with edge-lengths $\hat{\ell}(vw)$ that are approximations to the actual lengths $\ell(vw) = e^{\alpha f(vw)}$. We then show that computing these approximate edge-lengths does not change the asymptotic running times of our algorithms.

The first step is to note that in the proof of Lemma 4.2, we never used the fact that we reroute flow onto a *shortest* path. We only need that we reroute flow onto a *sufficiently short* path. More precisely, it is easy to convert the proof of Lemma 4.2 into a proof for the following claim.

LEMMA 4.9. *Suppose σ_i and τ satisfy the granularity condition, and let P be an ϵ-bad flow path of commodity i. Let Q be a path connecting the endpoints of P such that the length of Q*

is no more than $\epsilon'\ell(P)/2 + \epsilon'\frac{|f|}{D}|\ell|_1/2$ *greater than the length of the shortest path connecting the same endpoints. Then rerouting* σ_i *units of flow from path* P *to* Q *decreases* $|\ell|_1$ *by*

$$\Omega\left(\frac{\sigma_i}{\min\{D, kd(i)\}}|\ell|_1 \log m\right).$$

We will now show that in order to compute the lengths of paths up to the precision given in this lemma, we only need to compute the lengths of edges up to a reasonably small amount of precision.

By Lemma 4.9, the length of a path can have a rounding error of $\epsilon'\frac{|f|}{D}|\ell|_1/2$. Each path has at most n edges, so it will suffice to ensure that each edge has a rounding error of $\frac{1}{n}(\epsilon'\frac{|f|}{D}|\ell|_1/2)$. We will now bound this quantity. $|f|$ is the maximum flow on an edge and hence must be at least as large as the average flow on an edge, i.e., $|f| \geq \sum_{vw} f(vw)/m$. Every unit of flow contributes to the total flow on at least one edge, and hence $\sum_{vw} f(vw) \geq D$, and combining with the previous equation, we get that $|f|/D \geq 1/m$. $|\ell|_1$ is at least as big as the length of the longest edge, i.e., $|\ell|_1 \geq e^{\alpha|f|}$. Plugging in these bounds we see that it suffices to compute with an error of at most $(\epsilon'/nm)e^{\alpha|f|}$. Each edge has a positive length of at most $e^{\alpha|f|}$ and can be expressed as $e^{\alpha|f|}\rho$, where $0 < \rho \leq 1$. Thus we need to compute ρ up to an error of ϵ'/nm. To do so, we need to compute $O(\log(\epsilon^{-1}nm))$ bits, which by the assumption that ϵ is inverse polynomial in n is just $O(\log n)$ bits.

By using the Taylor series expansion of e^x, we can compute one bit of the length function in $O(1)$ time. Therefore, to compute the lengths of all edges at each iteration of REDUCE, we need $O(m \log n)$ time. In the deterministic implementation of REDUCE each iteration takes at least $\Omega(m \log n)$ time (the time required for cycle cancelling); therefore, the time spent on computing the lengths is dominated by the running time of an iteration.

The approximation above depends on the current value of $|f|$, which may change after each iteration. It was crucial that we recomputed the lengths of every edge in every iteration. The time to do so, $O(m \log n)$, would dominate the running time of the randomized implementation of REDUCE. (Recall that the randomized implementation does not do cycle cancelling.) Thus, we need to find an approximation that does not need to be recomputed at every iteration. We will choose one that does not depend on the current $|f|$ and hence will only need to be updated on the $O(n)$ edges on which the flow actually changes. We proceed to describe such an approximation that will depend on τ rather than $|f|$.

Throughout REDUCE all edge length are at most $e^{O(\alpha\tau)}$ and at least one edge has length more than $e^{\alpha\tau}$. Therefore, $|\ell|_1$ is at least $e^{\alpha\tau}$, and by the same argument as for the deterministic case $O(\epsilon^{-1} \log n)$ bits of precision suffice throughout REDUCE. When we first call REDUCE, we must spend $O(\epsilon^{-1}m \log n)$ time to compute all the edge lengths. For each subsequent iteration, we only need to spend $O(\epsilon^{-1}n \log n)$ time updating the $O(n)$ edges whose length have changed. Since each iteration of REDUCE is expected to take $O(\epsilon^{-1}(m + n \log n))$ time to compute shortest paths in FINDPATH, the time for updating edges is dominated by the time required by FINDPATH. While it appears that the time to compute initially all the edge lengths may dominate the time spent in one invocation of REDUCE, we shall see in §5 that whenever any of our algorithms calls REDUCE, it will have at least $\Omega(\log n)$ iterations. Each iteration is expected to take at least $\Omega(\epsilon^{-1}m)$ time to compute the shortest paths in FINDPATH. Therefore, the time spent on initializing lengths will be dominated by the running time of REDUCE.

Note that in describing the randomized version of FINDPATH in Lemma 4.6, we assumed we knew the exact lengths. However, by using the approximate lengths we do not significantly change a path's apparent contribution to the sum $\sum_i \sum_{P\in\mathcal{P}_i} \ell(P)f_i(P)$. Hence we do not significantly reduce the probability of selecting a bad path.

Thus we have shown that without any assumptions, REDUCE can be implemented deterministically in the same time as is stated in Theorem 4.8. Although for the randomized version, there is additional initialization time, for all the algorithms in this paper the initialization time is dominated by the time spent in the iterations of REDUCE.

THEOREM 4.10. *The times required for the deterministic implementations of procedure* REDUCE *stated in Theorem* 4.8 *hold without the assumption that exponentiation is a single step. The time required by the randomized implementations increases by an additive term of* $O(\epsilon^{-1}m \log n)$ *without this assumption.*

4.3.2. Further improvements for fixed ϵ. In this section we show how one can reduce the time per iteration of REDUCE for the case in which ϵ is a constant. First we show how using approximate lengths can reduce the time required by FINDPATH; we use an approximate shortest-paths algorithm that runs in $O(m + n\epsilon^{-1})$ time. Then we give improved implementation details for an iteration of REDUCE to decrease the time required by other parts of REDUCE.

We will describe how, given the lengths and an ϵ-bad path P from s to t, we can find a path Q with the same endpoints such that $\ell(Q) \leq \text{dist}_\ell(s,t) + \epsilon'\ell(P)/2$ in $O(m+n\epsilon^{-1})$ time. First, we discard all edges with length greater than $\ell(P)$, for they can never be in a path that is shorter than P (if P is a shortest path between s and t, then P is not an ϵ-bad path). Next, on the remaining graph, we compute shortest paths from s using approximate edge-lengths

$$\tilde{\ell}(v,w) = \frac{\epsilon'\ell(P)}{2n}\left\lceil \ell(vw)\frac{2n}{\epsilon'\ell(P)}\right\rceil,$$

thus giving us $\text{dist}_{\tilde{\ell}}(s,t)$, an approximation of $\text{dist}_\ell(s,t)$, the length of the actual shortest (s,t)-path. There are at most $n-1$ edges on any shortest path, and for each such edge, the approximate length is at most $\epsilon'\ell(P)/2n$ more than the actual length. Thus we know that

$$\text{dist}_{\tilde{\ell}}(s,t) \leq \text{dist}_\ell(s,t) + n\frac{\epsilon'\ell(P)}{2n} = \text{dist}_\ell(s,t) + \frac{\epsilon'\ell(P)}{2}$$

Further, since each shortest path length is an integer multiple of $\epsilon'\ell(P)/2n$ and no more than $\ell(P)$, we can use Dial's implementation of Dijkstra's algorithm [2] to compute $\text{dist}_{\tilde{\ell}}(s,t)$ in $O(m + n\epsilon^{-1})$ time.

Implementing FINDPATH with this approximate shortest path computation directly improves the time required by a deterministic implementation of REDUCE. The randomized implementation of FINDPATH with approximate shortest path computation requires $O(\epsilon^{-1}(m + n\epsilon^{-1}))$ expected time. In order to claim that an iteration of REDUCE can be implemented in the same amount of time, we must handle two difficulties: updating edge lengths and updating each edge's table of flow paths when flow is rerouted. Previously, these steps took $O(n \log n)$ time, which was dominated by the time for FINDPATH. We have reduced the time for FINDPATH, so the time for these steps now dominates. We show how to carry out these steps in $O(n)$ time. For the first step, we show that a table can be precomputed so that each edge length can be updated in constant time. For the second step, we sketch a three-level data structure that allows selection of a random flow path through an edge in $O(n)$ time and allows constant-time addition and deletion of flow paths.

Say that before computing the length $e^{\alpha f(vw)}$, we were to round $\alpha f(vw)$ to the nearest multiple of ϵ/c, for some constant c. This will introduce an additional multiplicative error of $1 + O(\epsilon/c)$ in the length of each edge and hence an additional multiplicative error of $1 + O(\epsilon/c)$ on each path. However, by arguments similar to the previous subsection, this will still give us a sufficiently precise approximation.

Now we will show that by rounding in this way there are a small enough number of possible values for $\ell(vw)$ that we can just compute them all at the beginning of an iteration of REDUCE and then compute the length of an edge by simply looking up the value in a precomputed table. The largest value of $\alpha f(vw)$ we will ever encounter is $O(\epsilon^{-1} \log n)$. Since we are only concerned with multiples of ϵ/c, there is a total of only $O(\epsilon^{-2} \log n)$ values that we will ever encounter. At the beginning of each iteration, we can compute each of these numbers to a precision of $O(\log n)$ bits in $O(\epsilon^{-2} \log^2 n)$ time. Once we have computed all these numbers, we can compute the length of an edge by computing $\alpha f(vw)$, truncating to a multiple of ϵ/c, and then looking up the value of $\ell(vw)$ in the table. This takes $O(1)$ time. Thus for constant ϵ, we are spending $O(\log^2 n + m) = O(m)$ time per iteration.

Now we address the problem of maintaining, for each edge, the flow paths going through that edge. Henceforth we will describe the data structure associated with a single edge. First suppose that all the flow paths carry the same amount of flow, i.e., σ_i is the same for each. In this case, we keep pointers to the flow paths in an array. We maintain that the array is at most one-quarter empty. It is then easy to select a flow path in constant expected time randomly; one randomly chooses an index and checks whether the corresponding array entry has a pointer to a flow path. If so, select that flow path. If not, try another index.

One can delete flow paths from the array in constant time. If one maintains a list of empty entries, one can also insert in constant time. If the array gets too full, copy its contents into a new array of twice the size. The time required for copying can be amortized over the time required for the insertions that filled the array. If the array gets too empty, copy its contents into a new array of half the size. The time required for copying can be amortized over the time required for the deletions that emptied the array. (See, for example, [1], for a detailed description of this data structure.)

Now we consider the more general case, in which the flow values of flow paths may vary. In this case, we use a three-level data structure. In the top level, the paths are organized according to their starting nodes. In the second level, the paths with a common starting node are organized according to their ending nodes. The paths with the same starting and ending nodes may be assumed to belong to the same commodity and hence all carry the same amount of flow. Thus these paths can be organized using the array as described above.

The first level consists of a list; each list item specifies a starting node, the total flow of all flow paths with that starting node, and a pointer to the second-level data structure organizing the flow paths with the given starting node. Each second-level data structure consists of a list; each list item specifies an ending node, the total flow of all flow paths with that ending node and the given starting node, and a pointer to the third-level data structure, the array containing flow paths with the given starting and ending nodes.

Now we analyze the time required to maintain this data structure. Adding and deleting a flow path takes constant time. Choosing a random flow path with the right probability can be accomplished in $O(n)$ time. First we randomly choose a value between 0 and the total flow through the edge. Then we scan the first-level list to select an appropriate item based on the value. Next we scan the second-level list pointed to by that item and select an item in the second-level list. Each of these two steps takes $O(n)$ time. Finally, we select an entry in the third-level array. In the third-level array, all the flows have the same σ_i; thus, this can be accomplished in $O(1)$ expected time by the scheme described above.

So we have shown that for constant ϵ, each of the three steps in procedure REDUCE can be implemented in $O(m)$ expected time, thus yielding the following theorem.

THEOREM 4.11. *Let $f = O(\tau)$ and τ and σ_i satisfy the granularity condition. Let*

$$H(k, d, \sigma) = \max_i \frac{\min\{D, kd(i)\}}{\sigma_i},$$

and let $\hat{k} = \min\{k, k^* \log d_{\max}\}$. *For any constant* $\epsilon > 0$ *the procedure* REDUCE *can be implemented in randomized* $O(m H(k, d, \sigma))$ *and in deterministic* $O(H(k, d, \sigma) m(\log n + \hat{k}))$ *time.*

5. Concurrent flow algorithms. In this section, we give approximation algorithms for the concurrent flow problem with uniform capacities. We describe two algorithms: CONCURRENT and SCALINGCONCURRENT. CONCURRENT is simpler and is best if ϵ is constant. SCALINGCONCURRENT gradually scales ϵ to the right value and is faster for small ϵ.

Algorithm CONCURRENT (see Fig. 2) consists of a sequence of calls to procedure REDUCE described in the previous section. The initial flow is constructed by routing each commodity i on a single flow path from s_i to t_i. Initially, we set $\sigma_i = d(i)$. Before each call to REDUCE we divide the flow quantum σ_i by 2 for every commodity where this is needed to satisfy the granularity condition (3). Each call to REDUCE modifies the multicommodity flow f so that either $|f|$ decreases by a factor of 2 or f becomes ϵ-optimal. (The procedure REDUCE can set a global flag to indicate whether it has concluded that f is ϵ-optimal.) In the latter case our algorithm can terminate and return the flow. As we will see, $O(\log m)$ calls to REDUCE will suffice to achieve ϵ-optimality.

```
CONCURRENT($G, \epsilon, \{d(i), (s_i, t_i) : 1 \le i \le k\}$)
For each commodity $i$: $\sigma_i \leftarrow d(i)$, create a simple path from $s_i$ to $t_i$ and route $d(i)$ flow on it.
$\tau \leftarrow |f|/2$.
While $f$ is not $\epsilon$-optimal,
  For every $i$,
    Until $\sigma_i$ and $\tau$ satisfy the granularity condition,
      $\sigma_i \leftarrow \sigma_i/2$.
  Call REDUCE ($f, \tau, \epsilon, d$).
  $\tau \leftarrow \tau/2$.
Return $f$.
```

FIG. 2. *Procedure* CONCURRENT.

THEOREM 5.1. *The algorithm* CONCURRENT *finds an* ϵ*-optimal multicommodity flow in* $O((\epsilon^{-1}k + \epsilon^{-3}m)(k^* n \log n + m(\log n + \min\{k, k^* \log d_{\max}\})) \log n)$ *or in expected time* $O((k\epsilon^{-2} + m\epsilon^{-4})(m + n \log n) \log n)$.

Proof. Immediately after the initialization we have $|f| \le D$. To bound the number of phases we need a lower bound on the minimum value of $|f|$. Observe that for every multicommodity flow f, the total amount of flow in the network is D. Every unit of flow contributes to the total flow on at least one of the edges, and hence $\sum_{vw \in E} f(vw) \ge D$. Therefore,

$$|f| \ge D/m. \tag{4}$$

This implies that the number of iterations of the main loop of CONCURRENT is at most $O(\log m)$. By Theorems 4.3 and 4.8, procedure REDUCE invoked during a single iteration of CONCURRENT first spends $O(m \log n)$ time initializing edge lengths and then executes

$$O\left(\epsilon^{-1} \max_i \frac{\min\{D, kd(i)\}}{\sigma_i}\right)$$

iterations. Throughout the algorithm for every i σ_i is either equal to $d(i)$ or is $\Theta(\epsilon^2 \tau / \log(m\epsilon^{-1}))$. In the first case,

$$\frac{\min\{D, kd(i)\}}{\sigma_i} = \frac{\min\{D, kd(i)\}}{d(i)} = \min\left\{\frac{D}{d(i)}, k\right\} \le k.$$

In the second case

$$\frac{\min\{D, kd(i)\}}{\sigma_i} = \min\{D, kd(i)\}\,\epsilon^{-2}\tau^{-1}\log(m\epsilon^{-1}) \le \epsilon^{-2}\frac{D}{\tau}\log(m\epsilon^{-1}).$$

Thus the total number of iterations of the loop of REDUCE is at most $O(\epsilon^{-1}(k + \epsilon^{-2}\frac{D}{\tau}\log(m\epsilon^{-1}))$, and the time spent on the initialization of the edge length is dominated. The value τ is halved at every iteration; therefore, the total number of calls required for all iterations is at most $O(\epsilon^{-1}k\log n)$ plus twice the number required for the last iteration of CONCURRENT. It follows from (4) that τ is $\Omega(\frac{D}{m})$, and the total number of iterations of the loop of REDUCE is at most $O(\epsilon^{-1}k\log n + \epsilon^{-3}m\log n)$. □

Consider the special case when ϵ is constant. We use the version of REDUCE implemented with an approximate shortest path computation and apply the bounds of Theorem 4.11 combined with a proof similar to that of Theorem 5.1 to get the following result:

THEOREM 5.2. *For any constant $\epsilon > 0$, an ϵ-optimal solution for the unit-capacity concurrent flow problem can be found in $O(m(k + m)\log^2 n)$ expected time by a randomized algorithm and in $O(m(k + m)(\log n + \min\{k, k^*\log d_{\max}\})\log n)$ time by a deterministic algorithm.*

If ϵ is less than a constant, we use the algorithm SCALINGCONCURRENT, shown in Fig. 3. It starts with a large ϵ and then gradually scales ϵ down to the required value. More precisely, algorithm SCALINGCONCURRENT starts by applying algorithm CONCURRENT with $\epsilon = \frac{1}{10}$. SCALINGCONCURRENT then repeatedly divides ϵ by a factor of 2 and calls REDUCE. After the initial call to CONCURRENT, f is $\frac{1}{10}$-optimal, i.e., $|f|$ is no more than twice the minimum possible value. Therefore, $|f|$ cannot be decreased below $\tau/2$, and every subsequent call to REDUCE returns an ϵ-optimal multicommodity flow (with the current value of ϵ). As in CONCURRENT, each call to REDUCE uses the largest flow quantum σ permitted by the granularity condition (3).

```
ScalingConcurrent($G$, $\epsilon'$, $\{d(i), (s_i, t_i) : 1 \le i \le k\}$)
$\epsilon \leftarrow \frac{1}{10}$.
Call Concurrent($G$, $\epsilon$, $\{d(i), (s_i, t_i) : 1 \le i \le k\}$), and let $f$ be the resulting flow.
  $\tau \leftarrow \tau/2$.

While $\epsilon > \epsilon'$,
  $\epsilon \leftarrow \epsilon/2$,
  For every $i$,
    Until $\sigma_i$ and $\tau$ satisfy the granularity condition,
      $\sigma_i \leftarrow \sigma_i/2$.
  Call Reduce ($f$, $\tau$, $\epsilon$, $\sigma$).
Return $f$.
```

FIG. 3. *Procedure* SCALING CONCURRENT.

THEOREM 5.3. *The algorithm* SCALINGCONCURRENT *finds an ϵ-optimal multicommodity flow in expected time $O((k\epsilon^{-1} + m\epsilon^{-3}\log n)(m + n\log n))$.*

Proof. As is stated in Theorem 5.2, the call to procedure CONCURRENT takes $O(km\log n + m^2\log m)$ time and returns a multicommodity flow f that is $\frac{1}{10}$-optimal; hence, $|f|$ is no more than twice the minimum. Therefore every subsequent call to REDUCE returns an ϵ-optimal multicommodity flow f.

The time required by one iteration is dominated by the call to REDUCE. The input flow f of REDUCE is 2ϵ-optimal, so, by Theorems 4.8 and 4.10, the time required by the randomized implementation of REDUCE is

$$O\left(\epsilon^{-1}(m + n\log n)\max_i \frac{\min\{D, kd(i)\}}{\sigma_i}\right).$$

We have seen that

$$\max_i \frac{\min\{D, kd(i)\}}{\sigma_i}$$

is at most $O(k + \epsilon^{-2}m\log m)$. The value of ϵ is reduced by a factor of two in every iteration. Therefore, the total time required for all iterations is at most twice the time required by the last iteration. The last iteration takes $O((k + \epsilon^{-2}m\log n)(\epsilon^{-1}(m + n\log n)))$ time, which proves the claim. □

Consider an implementation of CONCURRENT or SCALINGCONCURRENT with the deterministic version of REDUCE. The time required by FINDPATH does not depend on ϵ, so we cannot claim that the time is bounded by at most twice the time required for the last call to REDUCE. Since there are at most $\log\epsilon^{-1}$ iterations, we have the following theorem.

THEOREM 5.4. *An ϵ-optimal solution to the unit-capacity concurrent flow problem can be found deterministically in time $O(km\log^2 n + (k\log\epsilon^{-1} + \epsilon^{-2}m\log n)$ $(k^*n\log n + m(\log n + \min\{k, k^*\log d_{\max}\})))$.*

6. Two applications. In this section we describe two applications of our unit-capacity concurrent flow algorithm. The first application is to implement efficiently Leighton and Rao's sparsest cut approximation algorithm [14], and the second application is to minimize approximately channel width in VLSI routing; the second problem was considered by Raghavan and Thompson [16] and Raghavan [15].

We start by reviewing the result of Leighton and Rao concerning finding an approximately sparsest cut in a graph. For any partition of the nodes of a graph G into two sets A and B, the associated *cut* is the set of edges between A and B, and $\delta(A, B)$ denotes the number of edges in that cut. A cut is *sparsest* if $\delta(A, B)/(|A||B|)$ is minimized. Leighton and Rao [14] gave an $O(\log n)$-approximation algorithm for finding the sparsest cut of a graph. By applying this algorithm they obtained polylog-times-optimal approximation algorithms for a wide variety of NP-complete graph problems, including minimum feedback arc set, minimum cut linear arrangement, and minimum area layout.

Leighton and Rao exploited the following connection between sparsest cuts and concurrent flow. Consider an *all-pairs* multicommodity flow in G, where there is a unit of demand between every pair of nodes. In a feasible flow f, for any partition $A \cup B$ of the nodes of G, a total of at least $|A||B|$ units of flow must cross the cut between A and B. Consequently, one such edge must carry at least a $|A||B|/\delta(A, B)$ flow for the sparsest cut $A \cup B$. Leighton and Rao prove an approximate max-flow, min-cut theorem for the all-pairs concurrent flow problem, by showing that in fact this lower bound for $|f|$ is at most an $O(\log n)$ factor below the minimum value. Their approximate sparsest-cut algorithm makes use of this connection. More precisely given a nearly optimal length function (dual variables) they show how to find a partition $A \cup B$

that is within a factor of $O(\log n)$ of the minimum value of $|f|$ and, hence, of the value of the sparsest cut.[1]

The computational bottleneck of their method is solving a unit-capacity concurrent flow problem, in which there is a demand of 1 between every pair of nodes. In their paper, they appealed to the fact that concurrent flow can be formulated as a linear program and hence can be solved in polynomial time. A much more efficient approach is to use our unit-capacity approximation algorithm. The number of commodities required is $O(n^2)$. Leighton [12] has discovered a technique to reduce the number of commodities required. He shows that if the graph in which there is an edge connecting each source-sink pair is an expander graph, then the resulting flow problem suffices for the purpose of finding an approximately sparsest cut. (We call this graph the *demand graph.*) In an expander we have:

> For any partition of the node set into A and B, where $|A| \leq |B|$, the number of commodities crossing the associated cut is $\theta(|A|)$.

Therefore, the value of $|f|$ for this smaller flow problem is $\Omega(|A|/\delta(A, B))$. Since $|B| \geq n/2$, it follows that $n|f|$ is $\Omega(|A||B|)/\delta(A, B))$. The smaller flow problem essentially "simulates" the original all-pairs problem. Moreover, Leighton and Rao's sparsest-cut algorithm can start with the length function for the smaller flow problem in place of that for the all-pairs problem. Thus Leighton's idea allows one to find an approximate sparsest cut after solving a much smaller concurrent flow problem. If one is willing to tolerate a small probability of error in the approximation, one can use $O(n)$ randomly selected source-sink pairs for the commodities. It is well known how to select node pairs randomly so that, with high probability, the resulting demand graph is an expander.

By Theorem 5.2, algorithm CONCURRENT takes expected time $O(m^2 \log^2 m)$ to find an appropriate solution for this smaller problem.

THEOREM 6.1. *An $O(\log n)$-factor approximation to the sparsest cut in a graph can be found by a randomized algorithm in $O(m^2 \log^2 m)$ time.*

The second application we discuss is approximately minimizing channel width in VLSI routing. Often a VLSI design consists of a collection of modules separated by channels; the modules are connected by wires that are routed through the channels. For purposes of regularity the channels have uniform width. It is desirable to minimize that width in order to minimize the total area of the VLSI circuit. Raghavan and Thompson [16] give an approximation algorithm for minimizing the channel width. They model the problem as a graph problem in which one must route wires between pairs of nodes in a graph G so as to minimize the maximum number of wires routed through an edge. To solve the problem approximately, they first solve a concurrent flow problem where there is a commodity with demand 1 for each path that needs to be routed. An optimal solution f_{opt} fails to be a wire routing only in that it may consist of paths of *fractional* flow. However, the value of $|f_{\text{opt}}|$ is certainly a lower bound on the minimum channel width. Raghavan and Thompson give a randomized method for converting the fractional flow f_{opt} to an integral flow, increasing the channel width only slightly. The resulting wire routing f achieves channel width

$$|f| \leq |f_{\text{opt}}| + O(\sqrt{|f_{\text{opt}}| \log n}), \tag{5}$$

[1] Their algorithm also works for edge-weighted graphs; weights translate to edge capacities in the corresponding concurrent flow problem.

which is at most $w_{\min} + O(\sqrt{w_{\min}\log n})$, where $w_{\min}$ is the minimum width. In fact, the constant implicit in this bound is quite small. Later Raghavan [15] showed how this conversion method can be made deterministic.

The computational bottleneck is, once again, solving a unit-capacity concurrent flow problem. Theorems 5.3 and 5.4 are applicable and yield good algorithms. But if $w_{\min}$ is $\Omega(\log n)$, we can do substantially better.[2] In this case, a modified version of our algorithm ScalingConcurrent directly yields an integral f satisfying (5), although the big-Oh constant is not as good as that of [16].

Consider the procedure ScalingConcurrent. It consists of two parts. First the procedure Concurrent is called with $\epsilon = \frac{1}{10}$ to achieve $\frac{1}{10}$-optimality. Next, ScalingConcurrent repeatedly calls Reduce, reducing the error parameter ϵ by a factor of two every iteration, until the required accuracy is achieved. The demands are the same for every commodity; hence, σ_i is independent of i, and we shall denote it by σ.

We claim that if $w_{\min} = \Omega(\log n)$, then σ, which is initially 1 for this application, need never be reduced. Consequently, there remains a single path of flow per commodity, and the randomized conversion method of Raghavan and Thompson becomes unnecessary. We show that these paths constitute a routing with width $w_{\min} + O(\sqrt{w_{\min}\log n})$.

First suppose the call to Concurrent terminates because the granularity condition becomes false. At this point, we have that

$$1 > \epsilon^2\tau/(51\log(7m\epsilon^{-1})). \tag{6}$$

We have that $\tau \geq |f|/2$ and $\epsilon = \frac{1}{10}$, and therefore $|f| = O(\log n)$. By our assumption $w_{\min} = \Omega(\log n)$, and hence $|f| \leq w_{\min} + O(\sqrt{w_{\min}\log n})$.

Now assume that the call to Concurrent terminates with a $\frac{1}{10}$-optimal flow. We proceed with ScalingConcurrent. It terminates when the granularity condition becomes false, at which point inequality (6) implies that $\epsilon^2 = O((\log m)/\tau)$. The flow f is ϵ-optimal and integral. So $|f| \leq w_{\min} + O(w_{\min}\sqrt{(\log m)/\tau})$. Since $\tau = |f|/2 \geq w_{\min}/2$, this bound on $|f|$ is at most $w_{\min} + O(\sqrt{w_{\min}\log m})$, as required.

Theorem 6.2. *If $w_{\min}$ denotes the minimum possible width and $w_{\min} = \Omega(\log m)$, a routing of width $w_{\min} + O(\sqrt{w_{\min}\log n})$ can be found by a randomized algorithm in expected time $O(km\log n\log k + k^{3/2}(m + n\log n)/\sqrt{\log n})$ and by a deterministic algorithm in time $O(k\log k(k^*n\log n + mk^* + m\log n))$.*

Proof. We have shown that algorithm ScalingConcurrent finds the required routing if it is terminated as soon as the granularity condition becomes false with $\sigma = 1$. Now we analyze the time required.

We have $D = k$ and $d(i) = 1$ for every i, and throughout the algorithm we have $\sigma_i = 1$ for every i. The number of calls to Reduce during Concurrent is $O(\log k)$ (initially $|f| \leq k$, and it never gets below 1 with $\sigma = 1$). Therefore, the number iterations of the loop of Reduce required during Concurrent is $O(k\log k)$. Next we proceed with ScalingConcurrent. The number of iterations is at most $O(\log k)$, because ϵ is reduced by a factor of two each iteration, starts at $\frac{1}{10}$, and never gets below $1/k$. Each iteration is a call to Reduce, which in turn results in $O(k)$ iterations of the loop of Reduce.

The time required by one iteration of the loop deterministically is $O(k^*n\log n + m(k^* + \log m))$, and the total time to find a good routing of wires is $O(k\log k(k^*n\log n + mk^* + m\log n))$.

The expected time required by the randomized implementation of Reduce is $O(m\log n + k\epsilon^{-1}(m + n\log n))$. The total expected time required by Concurrent is $O(mk\log k\log n)$.

[2] This is the case of most interest, for if $w_{\min}$ is $O(\log n)$, then the error term in (5) dominates $w_{\min}$.

After the call to CONCURRENT ϵ decreases by a factor of two each iteration, it follows that the total expected time required for all iterations is $O(m \log n \log \epsilon^{-1})$ plus twice the time for the last call to REDUCE. During the last call to REDUCE, $\epsilon^{-1} = O(\sqrt{k/\log n})$, so the time required for all iterations is $O(km \log n \log k + k^{3/2}(m + n \log n)/\sqrt{\log n})$. This time dominates the time required by CONCURRENT since $w_{\min} = \Omega(\log n)$ implies $k = \Omega(\log n)$. □

Acknowledgments. We are grateful to Andrew Goldberg, Tom Leighton, Satish Rao, David Shmoys, and Pravin Vaidya for helpful discussions.

REFERENCES

[1] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press/McGraw-Hill, 1990, pp. 367–375.

[2] R. DIAL, *Algorithm 360: Shortest path forest with topological ordering*, Comm. Appl. Comput. Math., 12 (1969), pp. 632–633.

[3] M. L. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987) pp. 596–615.

[4] A. V. GOLDBERG, personal communication, Jan. 1991.

[5] A. V. GOLDBERG AND R. E. TARJAN, *Solving minimum-cost flow problems by successive approximation*, Math. Oper. Research, 15 (1990), pp. 430–466.

[6] M. D. GRIGORIADIS AND L. G. KHACHIYAN, *Fast approximation schemes for convex programs with many blocks and coupling constraints*, SIAM J. Optim., 4(1994), pp. 86–107.

[7] M. D. HANSEN, *Approximation algorithms for geometric embeddings in the plane with applications to parallel processing problems*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science, October 1989, IEEE, pp. 604–610.

[8] S. KAPOOR AND P. M. VAIDYA, *Fast algorithms for convex quadratic programming and multicommodity flows*, in Proceedings of the 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 147–159.

[9] P. KLEIN, A. AGRAWAL, R. RAVI, AND S. RAO, *Approximation through multicommodity flow*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, 1990, pp. 726–727.

[10] P. KLEIN AND C. STEIN, *Leighton-Rao might be practical: a faster approximation algorithm for uniform concurrent flow*, unpublished manuscript, November 1989.

[11] P. KLEIN, C. STEIN, AND É. TARDOS, *Leighton-Rao might be practical: faster approximation algorithms for concurrent flow with uniform capacities*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 1990, pp. 310–321.

[12] F. T. LEIGHTON, personal communication, November 1989.

[13] T. LEIGHTON, F. MAKEDON, S. PLOTKIN, C. STEIN, É. TARDOS, AND S. TRAGOUDAS, *Fast approximation algorithms for multicommodity flow problems*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991, pp. 101–111; J. Comput. System Sci., to appear.

[14] T. LEIGHTON AND S. RAO, *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms*, in Proceedings of the 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 422–431.

[15] P. RAGHAVAN, *Probabilistic construction of deterministic algorithms: approximating packing integer programs*, in Proceedings of the 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 10–18.

[16] P. RAGHAVAN AND C. D. THOMPSON, *Provably good routing in graphs: regular arrays*, in Proceedings of the 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 79–87.

[17] R. RAVI, A. AGRAWAL, AND P. KLEIN, *Ordering problems approximated: single-processsor scheduling and interval graph completion*, in Proceedings of the 18th International Conference on Automata, Languages, and Programming, Lecture Notes in Comput. Sci., vol. 510, Springer-Verlag, New York, 1991.

[18] F. SHAHROKHI AND D. W. MATULA, *The maximum concurrent flow problem*, J. Assoc. Comput. Mach., 37 (1990), pp. 318–334.

[19] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[20] E. TARDOS, *Improved approximation algorithm for concurrent multi-commodity flows*, Tech. report 872, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, October 1989.

[21] P. M. VAIDYA, *Speeding up linear programming using fast matrix multiplication*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science, 1989, pp. 332–337.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 488–509, June 1994

THE EXTENDED LOW HIERARCHY IS AN INFINITE HIERARCHY*

MING-JYE SHEU† AND TIMOTHY J. LONG‡

Abstract. Balcázar, Book, and Schöning introduced the extended low hierarchy based on the Σ-levels of the polynomial-time hierarchy as follows: for $k \geq 1$, level k of the extended low hierarchy is the set $EL_k^{P,\Sigma} = \{A \mid \Sigma_k^P(A) \subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT})\}$. Allender and Hemachandra and Long and Sheu introduced refinements of the extended low hierarchy based on the Δ- and Θ-levels, respectively, of the polynomial-time hierarchy: for $k \geq 2$, $EL_k^{P,\Delta} = \{A \mid \Delta_k^P(A) \subseteq \Delta_{k-1}^P(A \oplus \mathrm{SAT})\}$ and $EL_k^{P,\Theta} = \{A \mid \Theta_k^P(A) \subseteq \Theta_{k-1}^P(A \oplus \mathrm{SAT})\}$. This paper shows that the extended low hierarchy is properly infinite by showing, for $k \geq 2$, that $EL_k^{P,\Sigma} \subsetneqq EL_{k+1}^{P,\Theta} \subsetneqq EL_{k+1}^{P,\Delta} \subsetneqq EL_{k+1}^{P,\Sigma}$. The proofs use the circuit lower bound techniques of Håstad and Ko. As corollaries to the constructions, for $k \geq 2$, oracle sets B_k, C_k, and D_k, such that $\mathrm{PH}(B_k) = \Sigma_k^P(B_k) \supsetneqq \Delta_k^P(B_k)$, $\mathrm{PH}(C_k) = \Delta_k^P(C_k) \supsetneqq \Theta_k^P(C_k)$, and $\mathrm{PH}(D_k) = \Theta_k^P(D_k) \supsetneqq \Sigma_{k-1}^P(D_k)$ are obtained.

Key words. extended low hierarchy, circuit lower bounds, polynomial-time hierarchy, relativizations

AMS subject classifications. 68Q15, 68Q05

1. Introduction.

The low and high hierarchies in NP were introduced by Schöning in order to understand the internal structure of NP [Sch83]. The low hierarchy starts at P and grows "upward" toward the NP-complete sets, while the high hierarchy starts with the complete sets in NP and grows "downward" toward P. More formally, set A is in level k of the *low hierarchy* if $\Sigma_k^P(A) = \Sigma_k^P$, and A is in level k of the *high hierarchy* if $\Sigma_{k+1}^P = \Sigma_k^P(A)$. Many interesting sets have been located in these hierarchies: $\leq_T^P$-complete sets and $\leq_T^{SN}$-complete sets for NP form the first two levels of the high hierarchy, while sparse NP sets, NP sets with small circuits, and graph isomorphism have been placed in various levels of the low hierarchy.

These hierarchies were first extended to sets outside of NP by Balcázar, Book, and Schöning [BBS86]. In particular, they defined the *extended low hierarchy* as follows: set A is in level k of the extended low hierarchy if $\Sigma_k^P(A) \subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT})$. Once again, many interesting classes of sets have been located in the extended low hierarchy, such as sparse sets, sets with small circuits, and general left cuts of real numbers. Recently, refinements of the extended low hierarchy based on the Δ-classes and Θ-classes of the polynomial-time hierarchy were introduced by Allender and Hemachandra [AH92] and by Long and Sheu [LS91], respectively. Many of the sets previously located in the extended low hierarchy were more carefully and precisely located in new levels of the refined extended low hierarchy.

An interesting open question is whether the low hierarchy or the extended low hierarchy consists of an infinite number of distinct levels. This question is of interest for several reasons. If these hierarchies are properly infinite, then they provide a natural formal framework for categorizing and comparing the relative "power and complexity" of sets: sets at higher levels of the low hierarchy are more "complicated" than sets at lower levels. But more importantly, for the low hierarchy, if it contains at least two distinct levels, then $\mathrm{P} \neq \mathrm{NP}$. We also note that it is not even known if the low hierarchy is properly infinite under very natural assumptions, such as the polynomial-time hierarchy consisting of an infinite number of distinct levels.

For the extended low hierarchy, the situation is somewhat different. In proving optimal lower bounds for the extended low hierarchy, Allender and Hemachandra [AH92] constructed a sparse set that is not in $EL_2^{P,\Sigma}$, level two of the extended low hierarchy based on the Σ-levels

*Received by the editors August 26, 1991; accepted for publication (in revised form) December 21, 1992. This work was supported in part by National Science Foundation grant CCR-8909071.

†IBM Networking Software Laboratories, Research Triangle Park, North Carolina 27709.

‡Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210.

of the polynomial-time hierarchy. Since it is known that all sparse sets are in $EL_3^{P,\Theta}$, the third level of the extended low hierarchy based on the Θ-levels of the polynomial-time hierarchy, this shows that the extended low hierarchy has at least two distinct levels up through $EL_3^{P,\Theta}$. However, it was not known if the extended low hierarchy has distinct levels beyond $EL_3^{P,\Theta}$.

In this paper we show that the extended low hierarchy is in fact a properly infinite hierarchy. Specifically, we prove that

$$\forall k \geq 2,\ EL_k^{P,\Sigma} \subsetneq EL_{k+1}^{P,\Theta} \subsetneq EL_{k+1}^{P,\Delta} \subsetneq EL_{k+1}^{P,\Sigma},$$

where $EL_k^{P,\Sigma}$ ($EL_k^{P,\Delta}$, $EL_k^{P,\Theta}$) denotes level k of the extended low hierarchy based on the Σ-levels (Δ-levels, Θ-levels, respectively) of the polynomial-time hierarchy. This is the first example of a low hierarchy in complexity theory that has been shown to be properly infinite.

Our proofs follow the circuit lower bound techniques from Yao [Yao85], Håstad [Has87], and Ko [Ko89]. Of particular interest for us is Ko's result that there are sets relative to which the polynomial-time hierarchy collapses to any finite level:

$$\forall k \geq 1, \exists A\ [\mathrm{PH}(A) = \Sigma_k^P(A) \neq \Sigma_{k-1}^P(A)].$$

Note that A is in level $k+1$ of the extended low hierarchy based on Σ-levels of the polynomial-time hierarchy and is not likely to be in level k of the extended low hirarchy. The condition for A not being in level k of the extended low hierarchy is as follows:

$$\mathrm{PH}(A) = \Sigma_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT}).$$

Thus, a direct modification of Ko's proofs yields that every Σ-level of the extended low hierarchy is distinct. In this paper we are able to extend Ko's collapsing results to Δ- and Θ-levels of relativized polynomial-time hierarchies.

In §3, we extend Yao and Håstad's separation results by showing the existence of oracle sets A, B, and C such that for all $k \geq 2$, $\Delta_k^P(A) \neq \Sigma_k^P(A)$, $\Theta_k^P(B) \neq \Delta_k^P(B)$, and $\Sigma_{k-1}^P(C) \neq \Theta_k^P(C)$. In order to obtain our results, we introduce, for the first time, circuit models for the Δ- and Θ-classes of relativized polynomial-time hierarchies. In §4, we combine the separation results of §3 with encoding techniques to construct oracles D_k, E_k, and F_k such that

$$\forall k \geq 1\ \exists D_k\ [\mathrm{PH}(D_k) = \Sigma_k^P(D_k) \not\subseteq \Delta_k^P(D_k \oplus \mathrm{SAT})],$$

$$\forall k \geq 2\ \exists E_k\ [\mathrm{PH}(E_k) = \Delta_k^P(E_k) \not\subseteq \Theta_k^P(E_k \oplus \mathrm{SAT})],$$

and

$$\forall k \geq 2\ \exists F_k\ [\mathrm{PH}(F_k) = \Theta_k^P(F_k) \not\subseteq \Sigma_{k-1}^P(F_k \oplus \mathrm{SAT})].$$

These results show that every level of the extended low hierarchy is distinct. Immediate corollaries are the existence of oracles G_k, H_k, and I_k, for all $k \geq 2$, such that $\mathrm{PH}(G_k) = \Sigma_k^P(G_k) \neq \Delta_k^P(G_k)$, $\mathrm{PH}(H_k) = \Delta_k^P(H_k) \neq \Theta_k^P(H_k)$, and $\mathrm{PH}(I_k) = \Theta_k^P(I_k) \neq \Sigma_{k-1}^P(I_k)$. These corollaries answer open questions posed by Ko [Ko89].

2. Preliminaries.

2.1. Basic definitions and notation.

Σ denotes an arbitrary alphabet of size at least two. For $x \in \Sigma^*$, $|x|$ denotes the length of x, with λ denoting the string of length 0. For a set A over Σ, $\overline{A}$ denotes its complement $\Sigma^* - A$, and for a class of sets $\mathcal{C}$, co-$\mathcal{C}$ denotes the set of complements of the elements of $\mathcal{C}$. For arbitrary sets A and B, $A \oplus B = \{0x \mid x \in A\} \cup \{1x \mid$

$x \in B\}$. Let $\| A \|$ denote the cardinality of a set A, and let $\| A^{\leq n} \|$ denote the cardinality of $A^{\leq n}$; that is, the cardinality of A restricted to strings of length at most n.

$L(M)$ denotes the set accepted by Turing machine M, and $L(M, A)$ denotes the set accepted by an oracle Turing machine M using oracle set A. The classes P and NP have their standard definitions. P(A) and NP(A) are the A-relativized versions of P and NP. For each oracle set A, we define the set $K(A)$ as follows: for a nondeterministic oracle machine M_i, a string x, and an integer j, let $\langle i, x, 1^j \rangle \in K(A)$ if and only if the nondeterministic machine M_i, using A as the oracle, accepts x in at most j steps. It is well known that, for every set A, $K(A)$ is $\leq_m^P$-complete for NP(A).

2.2. Polynomial-time hierarchy. The classes of the polynomial-time hierarchy are $\{\Sigma_k^P, \Pi_k^P, \Delta_k^P, \Theta_k^P \mid k \geq 0\}$, where

$$\Sigma_0^P = \Pi_0^P = \Delta_0^P = \Theta_0^P = \text{P}, \text{ and for } k \geq 0,$$

$$\Sigma_{k+1}^P = \text{NP}(\Sigma_k^P),$$

$$\Pi_{k+1}^P = \text{co-}\Sigma_{k+1}^P, \Delta_{k+1}^P = \text{P}(\Sigma_k^P),$$

and

$$\Theta_{k+1}^P = \Theta(\Sigma_k^P).^1$$

Sets in the polynomial-time hierarchy relative to A have also been characterized in terms of polynomial bounded quantification over predicates in P(A) [Sto76], [Wra76]. We adopt the following definitions from [Ko89]. A predicate σ of a set variable A and a string variable x is called a polynomial-time predicate, or a P^1-predicate, if there exist an oracle machine M and a polynomial p such that p bounds the running time of M, and for all sets A and all strings x, M^A accepts x if and only if $\sigma(A; x)$ is true. Thus, a set B is in P(A) if and only if there exists a P^1-predicate σ such that for all x, $x \in B \Leftrightarrow \sigma(A; x)$ holds.

Define the $\Sigma_0^{P,1}$-predicates to be the P^1-predicates. For each $k \geq 1$, define a $\Sigma_k^{P,1}$-predicate to be any predicate of the form:

$$(\exists y_1, |y_1| \leq p(|x|))(\forall y_2, |y_2| \leq p(|x|)) \cdots (Q_k y_k, |y_k| \leq p(|x|))\, \sigma(A; \langle x, y_1, \ldots, y_k \rangle),$$

where σ is a P^1-predicate, p is a polynomial, and Q_k is $\exists$ if k is odd and Q_k is $\forall$ if k is even. Without loss of generality, we assume that p also bounds the running time of the oracle machine M that decides σ with oracle A. The classes of the relativized polynomial-time hierarchy (PH(A)) can be defined as follows:

$$\Sigma_k^P(A) = \{L \mid (\exists \Sigma_k^{P,1}\text{-predicate } \tau)[x \in L \Leftrightarrow \tau(A; x)]\},$$

$$\Pi_k^P(A) = \text{co-}\Sigma_k^P(A),$$

$$\Delta_{k+1}^P(A) = \text{P}(\Sigma_k^P(A)),$$

and

$$\Theta_{k+1}^P(A) = \Theta(\Sigma_k^P(A)), \text{ for } k \geq 0.$$

It is known that for each level of the relativized polynomial-time hierarchy, there is a $\Sigma_k^P(A)$-complete set. These complete sets can be defined inductively as follows: Let $K^1(A) = K(A)$ and $K^k(A) = K(K^{k-1}(A))$ for $k > 1$. Then for each $k \geq 1$ and each set A, $K^k(A)$ is complete for $\Sigma_k^P(A)$.

[1] The Θ operator is defined as follows: for any class C, a set $L \in \Theta(C)$ if and only if there are a deterministic polynomial time oracle machine M and a set $A \in C$ such that $L = L(M, A)$ and, on any input x, M makes at most $O(\log |x|)$ queries to A.

2.3. Circuits. A circuit is a rooted tree, where each nonleaf node is associated with a gate. In this paper, we use four types of gates: AND, OR, MaxOdd, and XOR. For a MaxOdd gate, let its input lines from left to right be $b_0, b_1, \ldots, b_n$. A MaxOdd gate outputs 1 if and only if i is odd, where b_i is the rightmost input line with input 1. A MaxOdd gate outputs 0 if all the inputs are 0. An XOR gate outputs 1 if and only if the number of its 1-inputs is odd. The fanins of all the gates are unlimited. Each leaf of a circuit is associated with a constant 0, a constant 1, a variable x, or a negated variable $\overline{x}$. The *size* of a circuit C is defined to be the number of gates in C. Given a circuit C with only AND and OR gates, its dual circuit $\hat{C}$ is defined inductively as follows: the dual of a constant or a variable is its negation, and the dual of a circuit C that is an OR (AND) of n children C_i, $1 \leq i \leq n$, is the AND (OR) of $\hat{C}_i$, $1 \leq i \leq n$.

A circuit computes a function on its variables in the obvious way. Each variable is represented by v_z for some string $z \in \Sigma^*$. Let V be the set of variables in a circuit C. A *restriction* ρ of a circuit C is a mapping from V to $\{0, 1, *\}$. Let $C\lceil_\rho$ denote the circuit C' obtained from C by replacing each variable x with $\rho(x) = 0$ by 0 and each y with $\rho(y) = 1$ by 1. If a restriction ρ maps no variables to $*$, we say ρ is a complete assignment for the variables of C. Also, we say a restriction ρ *completely determines* C if $C\lceil\rho$ computes a constant function 0 or 1.[2] For any set $A \subseteq \Sigma^*$, the natural assignment ρ_A on all variables v_z, $z \in \Sigma^*$ is: $\rho_A(v_z) = 1$ if $z \in A$ and $\rho_A(v_z) = 0$ if $z \notin A$. For any pair of sets B_1, B_0, with $B_1 \cap B_0 = \emptyset$, the assignment ρ_{B_1,B_0} on all variables v_z, $z \in \Sigma^*$, is $\rho_{B_1,B_0}(v_z) = 1$, if $z \in B_1$, $\rho_{B_1,B_0}(v_z) = 0$, if $z \in B_0$, and $\rho_{B_1,B_0}(v_z) = *$, if $z \notin B_1 \cup B_0$.

Note that applying a restriction ρ to a circuit C and its dual circuit $\hat{C}$ results in another pair of dual circuits. This can be expressed by the following proposition.

PROPOSITION 2.1. *Let C be a circuit with only AND and OR gates, and let $\hat{C}$ be the dual circuit of C. Let ρ be a restriction of C. Then $\hat{C}\lceil_\rho$ is the dual circuit of $C\lceil_\rho$.*

Random restrictions of circuits were first introduced by Furst, Saxe, and Sipser [FSS81]. We use two probability spaces of restrictions $R^+_{q,\mathcal{B}}$ and $R^-_{q,\mathcal{B}}$ from [Has87], as well as the restriction $g(\rho)$, where $\rho \in R^+_{q,\mathcal{B}}$ or $R^-_{q,\mathcal{B}}$. Let $\mathcal{B} = \{B_j\}_{j=1}^r$ be a partition of variables in a circuit C, and let q be a value between 0 and 1. To define a random restriction ρ in $R^+_{q,\mathcal{B}}$, for each B_j, $1 \leq j \leq r$, let $s_j = *$ with probability q and let $s_j = 0$ with probability $1 - q$. Then, independently, for each variable $x \in B_j$, let $\rho(x) = s_j$ with probability q and $\rho(x) = 1$ with probability $1 - q$. Next, for each $\rho \in R^+_{q,\mathcal{B}}$, define a restriction $g(\rho)$ such that for all B_j with $s_j = *$, let V_j be the set of all variables in B_j that are given value $*$ by ρ. $g(\rho)$ selects one variable y in V_j and gives value $*$ to y and value 1 to all other variables in V_j. The probability space $R^-_{q,\mathcal{B}}$ and $g(\rho)$ are defined analogously by interchanging the roles of 0 and 1.

Circuits can be used to describe the computation of oracle machines on input strings [FSS81]. The following proposition is taken from Ko [Ko89].

PROPOSITION 2.2 [Ko89]. *Let M be a deterministic oracle machine with runtime $\leq p(n)$, where p is a polynomial. Then, for each x, there is a depth-2 circuit $C = C_{M,x}$ satisfying the following properties*:

(1) *C is an* OR *of* AND*s*;

(2) *the top fanin of C is $\leq 2^{p(|x|)}$ and the bottom fanin of C is $\leq p(|x|)$; and*

(3) *for any set A, $C\lceil_{\rho_A} = 1$ if and only if $M^A(x)$ accepts.*

Intuitively, each AND gate of C represents one of the accepting computation paths of M on input x, where the variables of the AND gate represent the query strings and answers of that path. It is not difficult to see that the above proposition also holds if we require that the circuit C be an AND of ORs.[3]

[2] Note that a restriction that is not a complete assignment may still completely determine C.

[3] Since M is a deterministic oracle machine, there is a determinisitc oracle machine M' such that M' behaves exactly like M except reversing accept/reject at the end of the computation. The desired AND of ORs circuit is the dual circuit of the circuit obtained from M' by Proposition 2.2.

We call a depth-$(k+1)$ circuit a Σ_k-circuit if it has alternating OR and AND gates, starting with a top OR gate. A $\Sigma_k(m)$-circuit (Fig. 1) is a Σ_k-circuit such that

(1) the number of gates at level 1 to level $k-1$ is bounded by 2^m,

and

(2) fanins at level $k+1$ are $\leq m$.

Note that there is no restriction in fanin at level k. A depth-$(k+1)$ circuit is a Π_k-circuit ($\Pi_k(m)$-circuit) if its dual circuit is a Σ_k-circuit ($\Sigma_k(m)$-circuit). Now we can translate $\Sigma_k^{P,1}$-predicates to circuits.

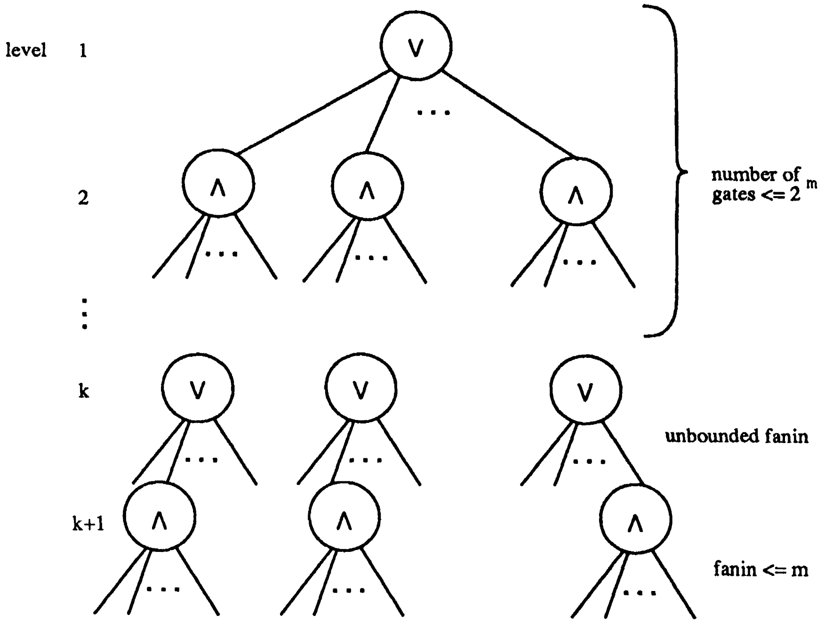


FIG. 1. *A $\Sigma_k(m)$-circuit.*

PROPOSITION 2.3 [Ko89]. *Let $k \geq 1$. For every $\Sigma_k^{P,1}$-predicate τ there is a polynomial q such that for every x, there exists a $\Sigma_k(q(|x|))$-circuit $C_{\tau,x}$, having the property that for any set A, $C_{\tau,x}\lceil_{\rho_A} = 1$ if and only if $\tau(A; x)$ is true. Also, for each variable v_z ($\overline{v_z}$) in $C_{\tau,x}$, $|z| \leq q(|x|)$.*

Sipser [Sip83] introduced a family of functions f_k^n, $n \geq 1$, $k \geq 1$, computed by special types of circuits. Håstad later introduced a modified family of f_k^n functions [Has87], and Ko, still later, introduced a further modification of the f_k^n functions in constructing oracles that separate and collapse relativized polynomial-time hierarchies [Ko89]. We will use Ko's defnition of f_k^n functions as well as introducing similar functions that will be used in our separation results in the next section. These circuits and their functions are pictured in Figs. 2, 3, and 4.

DEFINITION 2.4 [Ko89]. *For each $n \geq 1$ and $k \geq 1$, a C_k^n circuit is a depth-k circuit such that*:

(1) *C_k^n has alternating OR and AND gates, with a top OR gate*;

(2) *all the fanins of C_k^n are exactly n, except the bottom fanins, which are exactly $\sqrt{n}$*; *and*

(3) *each leaf of C_k^n is a unique positive variable.*

We let f_k^n denote the function computed by C_k^n.

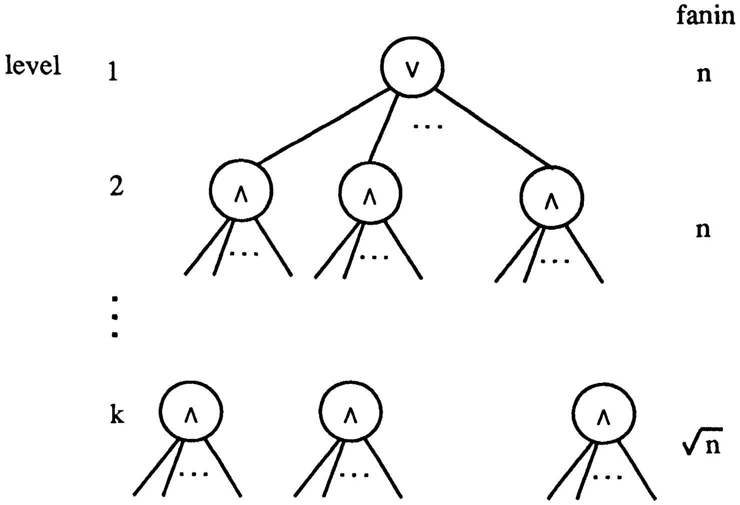


FIG. 2. *A C_k^n circuit.*

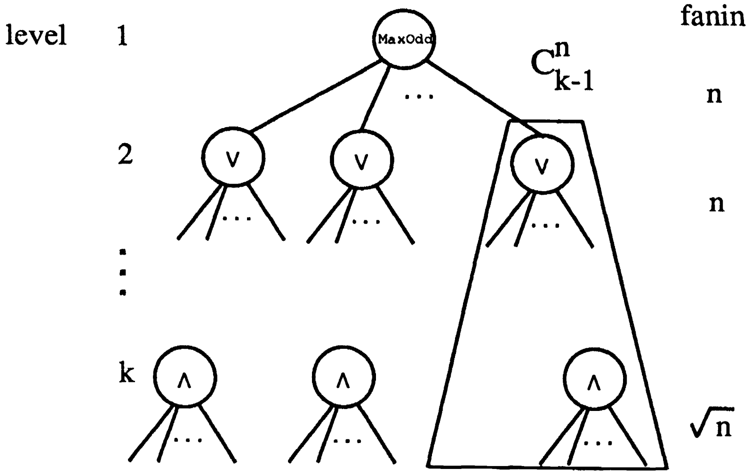


FIG. 3. *An H_k^n circuit.*

DEFINITION 2.5. (1) *For each $n \geq 1$ and $k \geq 1$, an H_k^n circuit is a depth-k circuit with a top* MaxOdd *gate having fanin exactly n such that each subcircuit of the top* MaxOdd *gate is a C_{k-1}^n circuit computing an f_{k-1}^n function and that all the variables in H_k^n are unique. We let h_k^n denote the function computed by H_k^n.*

(2) *For each $n \geq 1$ and $k \geq 1$, a D_k^n circuit is a depth-k circuit with a top* XOR *gate having fanin exactly* 2 *such that each subcircuit of the top* XOR *gate is a C_{k-1}^n circuit computing an f_{k-1}^n function and that all the variables in D_k^n are unique.*

We let l_k^n denote the function computed by D_k^n.

Before ending this section, we state some important lemmas, including the switching lemma on constant-depth circuits from Håstad [Has87].

LEMMA 2.6 (Switching Lemma) [Has87]. *Let G be an* AND *of* OR*s with bottom fanin $\leq t$, and let $\mathcal{B} = \{B_j\}$ be a partition of the variables in G. Then, for a random restriction ρ*

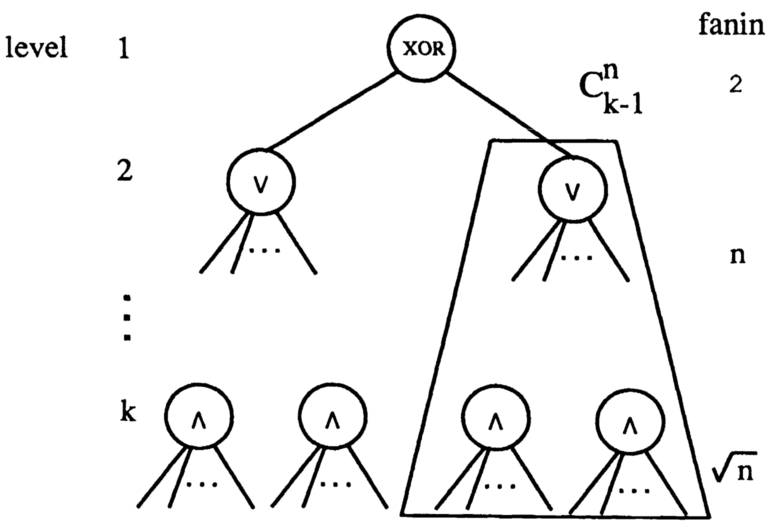


FIG. 4. *A D^n_k circuit.*

from $R^+_{q,\mathcal{B}}$, $Pr(G\lceil_{\rho g(\rho)}$ is not equivalent to a circuit of OR *of* AND*s with bottom fanin $\le s$) is bounded by α^s, where $\alpha < 6qt$. The above probability also holds with $R^+_{q,\mathcal{B}}$ replaced by $R^-_{q,\mathcal{B}}$ or with G being an* OR *of* AND*s circuit to be converted to an* AND *of* OR*s circuits.*

The next lemma is a generalized form of the switching lemma.

LEMMA 2.7 [Has87, Ko89]. *Let $\{C_i\}^n_{i=1}$ be a collection depth-*2 AND-of-OR *and* OR-of-AND *circuits, where the bottom fanin of each C_i is $\le t$. Let V be the set of the variables in $\{C_i\}^n_{i=1}$, and let $\mathcal{B} = \{B_j\}$ be a partition of the variables in V. q is a number between* 0 *and* 1 *and ρ is a random restriction from $R^+_{q,\mathcal{B}}$ or $R^-_{q,\mathcal{B}}$. The probability that every $C_i\lceil_{\rho g(\rho)}$, which is an* AND-of-OR *or* OR-of-AND, *can be converted to a* OR-of-AND *or* AND-of-OR, *respectively, with bottom fanin $< s$ is greater than* 2/3, *if $n \le 1/(3\alpha^s)$, where $\alpha < 6qt$.*

The importance of the switching lemma is that a depth-$(k + 1)$, $k \ge 2$, alternating OR-AND-gate circuit, with small bottom fanins, can be converted to an equivalent depth-k alternating OR-AND-gate circuit, with small bottom fanins. This "shrinking" effect is done by converting the bottom 2-level OR-of-AND (AND-of-OR) subcircuits to AND-of-OR (OR-of-AND) subcircuits and then collapsing the AND (OR) gates at level k and $k - 1$ of the circuit. Our next lemma is a formal statement of this shrinking effect.

LEMMA 2.8. *For each $k \ge 2$, let $\{C_i\}^n_{i=1}$ be a collection of $\Sigma_k(\gamma)$-circuits and $\Pi_k(\gamma)$-circuits. Let V be the set of the variables in $\{C_i\}^n_{i=1}$, and let $\mathcal{B} = \{B_j\}$ be a partition of the variables in V. Let q be a number between* 0 *and* 1, *and let ρ be a random restriction from $R^+_{q,\mathcal{B}}$ or $R^-_{q,\mathcal{B}}$. The probability that every $\Sigma_k(\gamma)$-circuit and every $\Pi_k(\gamma)$-circuit, after applying the random restriction ρ, can be converted to an equivalent $\Sigma_{k-1}(\gamma)$-circuit and an equivalent $\Pi_{k-1}(\gamma)$-circuit is greater than* 2/3, *if $n \le 1/(3(2\alpha)^\gamma)$ where $\alpha < 6q\gamma$.*

Lemmas 2.9 and 2.10 show the effects of a random restriction $\rho \in R^+_{q,\mathcal{B}}$ or $R^-_{q,\mathcal{B}}$ on the circuits that compute f^n_k, h^n_k, and l^n_k functions. The effect of random restrictions on the circuits computing f^n_k functions is due to Håstad [Has87]. Lemma 2.9 shows that a random restriction $\rho \in R^+_{q,\mathcal{B}}$ (or $R^-_{q,\mathcal{B}}$, depending on whether k is odd or even) transforms a circuit computing f^n_k (h^n_{k+1}, or l^n_{k+1}), to a circuit computing a function very close to f^n_{k-1} (h^n_k, or l^n_k). Lemma 2.10 is a stronger form of Lemma 2.9. All proofs follow easily from the proofs in [Ko89].

LEMMA 2.9. *For each $k \ge 2$, there exists an integer n_k such that the following hold for all*

$n \geq n_k$: Let C be a C_k^n circuit computing an f_k^n function (an H_{k+1}^n circuit computing an h_{k+1}^n function, or a D_{k+1}^n circuit computing an l_{k+1}^n function), let $q = n^{-1/3}$, and let ρ be a random restriction from $R_{q,\mathcal{B}}^+$ if k is even or from $R_{q,\mathcal{B}}^-$ if k is odd. The probability that $C\lceil_{\rho g(\rho)}$ contains a subcircuit computing an f_{k-1}^n function (an h_k^n function, or an l_k^n function, respectively) is greater than $2/3$.

LEMMA 2.10. *For each $k \geq 2$, let $\{C_i\}_{i=1}^t$ be t C_k^n circuits, each computing a function f_k^n, with pairwise disjoint variables. Let $q = n^{-1/3}$. If $t \leq 2^{\delta n^{1/6}}$ with $\delta < 1$, then for sufficiently large n, there exists a restriction ρ from $R_{q,\mathcal{B}}^+$ if k is even or from $R_{q,\mathcal{B}}^-$ if k is odd, such that every $C_i\lceil_{\rho g(\rho)}$, $1 \leq i \leq t$, contains a subcircuit that computes an f_{k-1}^n function. This is also holds for H_{k+1}^n and D_{k+1}^n circuits and h_{k+1}^n and l_{k+1}^n functions.*

3. Separation results for relativized polynomial-time hierarchies. Yao and Hastad [Yao85], [Has87] proved the existence of oracles, relative to which, the polynomial-time hierarchy has an infinite number of distinct levels. In this section, we further separate the Σ-, Δ-, and Θ-classes of relativized polynomial-time hierarchies. Our separating results show that there is an oracle A such that for all $k \geq 2$, $\Sigma_{k-1}^P(A) \subsetneq \Theta_k^P(A) \subsetneq \Delta_k^P(A) \subsetneq \Sigma_k^P(A)$. Although Yao and Hastad's results imply part of our separation results, we show different proofs here and will make use of them in the later sections.

3.1. Separating Δ-classes and Σ-classes in relativized polynomial-time hierarchies. In this section, we show that there exists a recursive oracle A such that for all $k \geq 2$, $\Sigma_k^P(A) \neq \Delta_k^P(A)$. We will find, for each $k \geq 2$, a set $L_k^A \in \Sigma_k^P(A)$ such that $L_k^A \notin \Delta_k^P(A)$. By dovetailing the diagonalization for each $k > 0$, the oracle set A can be constructed so that $\Delta_k^P(A) \subsetneq \Sigma_k^P(A)$ for all $k > 0$. Note that when $k = 1$, separation of $\Sigma_1^P(A)$ and $\Delta_1^P(A)$ was already done in [BGS75].

In order to use the circuit lower bound techniques, we need to define circuits for $\Delta_k^P(A)$. Consider a deterministic polynomial-time bounded oracle machine M using oracle set B. On any input x, we can perform computations of M on x. By Proposition 2.2, there is a two-level OR-of-ANDs circuit $C_{M,x}$ such that $x \in L(M, B)$ if and only if $C_{M,x}\lceil_{\rho_B} = 1$. Each AND gate of $C_{M,x}$ represents an accepting computation path of T, relative to some oracle. The fanins (variables) of each AND gate are the query strings in the corresponding computation path of T. For the convenience of discussion, we will label each AND gate with $p(|x|)$ bits, where p is the polynomial bound of the running time of M on any input x. For each label $l = b_1 \ldots b_{p(|x|)}$, b_i represents the answer of the ith query string in the corresponding computation path: $b_i = 1$ if and only if the answer to the ith query string is YES. It is not difficult to see that every two labels are distinct and that for any two labels l and l' with same prefix $b_1 \ldots b_i$, $1 \leq i < p(|x|)$, the corresponding computation paths π_l and $\pi_{l'}$ have the same query strings $q_1, \ldots, q_{i+1}$.

Now if we replace B by $K^{k-1}(A)$, then the language accepted by M is in $\Delta_k^P(A)$. As stated in §2, for each variable z of the form $\langle i, y, 1^j \rangle$, there are a polynomial q and a $\Sigma_{k-1}(q(|z|))$-circuit $C_{K^{k-1},z}$ such that $z \in K^{k-1}(A)$ if and only if $C_{K^{k-1},z}\lceil_{\rho_A} = 1$. We then replace each positive variable v_z of $C_{M,x}$ by $C_{K^{k-1},z}$ if z is of the correct syntactic form or by 0 if z is not of the correct syntactic form. For each negative variable v_z of $C_{M,x}$, we replace it by the dual circuit of $C_{K^{k-1},z}$ if z is of the correct syntactic form or by 1 if z is not of the correct syntactic form. The entire circuit consists of $k+2$ levels. Now we formally define, for $k \geq 2$, $\Delta_k(m)$-circuits.

DEFINITION 3.1. *For $k \geq 2$, C is a $\Delta_k(m)$-circuit if and only if*

(1) *C has $k + 2$ levels; and the top gate is an* OR *with fanin $\leq 2^m$;*

(2) *the gates on the second level are* AND *gates with fanins $\leq m$;*

(3) *each* AND *gate G_j at the second level has a distinct label of m bits, $b_1 \ldots b_m$, and if $b_i = 1$ then the ith subcircuit from the left of G_j is a $\Sigma_{k-1}(m)$-circuit, and if $b_i = 0$ then the ith subcircuit from the left of G_j is a $\Pi_{k-1}(m)$-circuit;*

(4) *for any two* AND *gates G_j and G_n, labeled l_j and l_n, at the second level, if the first r bits, $r < m$, of labels l_j and l_n are the same, then the first r subcircuits from the left of G_j and G_n are the same*; *and*

(5) *for any two* AND *gates G_j and G_n, labeled l_j and l_n, at the second level, if the first bit from the left of l_j and l_n, where l_j and l_n differ, is b_r, $r \leq m$, then the rth subcircuits from the left of G_j and G_n are dual circuits.*

The following lemma states that for all $k \geq 2$ and for any arbitrary set A, if $L \in \Delta_k^P(A)$, then there is a Δ_k-circuit C such that $x \in L \Leftrightarrow C\lceil_{\rho_A} = 1$.

LEMMA 3.2. *Let $k \geq 2$. For every set $L \in \Delta_k^P(A)$, there is a polynomial q such that for every input string x, there is a corresponding $\Delta_k(q(|x|))$-circuit $C_{L,x}$ such that $x \in L \Leftrightarrow C_{L,x}\lceil_{\rho_A} = 1$.*

Our goal is, for every $k \geq 2$, to find a set A such that $\Sigma_k^P(A)$ properly contains $\Delta_k^P(A)$. First, for fixed $k \geq 1$, define the language

$$L_k^A = \{x0^{kn} \mid |x| = 2n, (\exists y_1, |y_1| = n)(\forall y_2, |y_2| = n) \cdots (Q_k y_k, |y_k| = n) 0xy_1y_2 \ldots y_k \in A\},$$

where Q_k is $\exists$ if k is odd and Q_k is $\forall$ if k is even. It is clear that $L_A \in \Sigma_k^P(A)$.

Let $M_1, M_2, \ldots$ be an effective enumeration of deterministic polynomial-time bounded oracle machines with running time bounded by polynomials $p_1, p_2, \ldots$, respectively. The construction of set A is by stages. Let $t(n)$ denote the maximum length of strings in A or $\overline{A}$ decided by stage n. Let $D(n)$, the diagonalization area for stage n, be the set $\{x \in \Sigma^* \mid t(n-1) + 1 \leq |x| \leq t(n)\}$. At stage n of the construction, we will find a witness 0^m, m is a multiple of $k+2$, such that, $0^m \in L_k^A \Leftrightarrow 0^m \notin L(M_n, K^{k-1}(A))$; that is, at each stage $n > 0$, we want to satisfy the following requirement R_n:

$$R_n : (\exists x = 0^m)(\exists B \subseteq D(n))[0^m \in L_k^B \Leftrightarrow 0^m \notin L(M_n, K^{k-1}(A'))],$$

where $A' = A(n-1) \cup B$ and $A(n-1)$ denotes the strings in A decided prior to stage n. Let C_0 be the obvious k-level circuit for the predicate $0^m \in L_k^B$. C_0 contains a subcircuit that computes a $f_k^{2^{m/(k+2)}}$ function. By Lemma 3.2, there exist a polynomial q and a $\Delta_k(q(m))$-circuit C_n that corresponds to the computation of the machine M_n on input 0^m with oracle $K^{k-1}(A)$. Since the length of query strings to A in the circuit C_n is bounded by $q(m)$, we let $t(n) = q(m)$. The requirement R_n can then be rewritten as

$$R'_n : (\exists x = 0^m)(\exists B \subseteq D(n))[C_0\lceil_{\rho_B} = 1 \Leftrightarrow C_n\lceil_{\rho_{A'}} = 0],$$

where $A' = A(n-1) \cup B$. Now let $s_k(v)$ be the minimum r such that a $\Delta_k(r)$-circuit can compute an f_k^v function. That requirement R'_n can be satisfied follows from the following lower bound on $s_k(v)$.

THEOREM 3.3. *For all $k \geq 2$, for sufficiently large v, $s_k(v) \geq \delta v^{1/5}$, where $\delta = 1/48$.*

Proof. The proof is by induction on k. For the base case of $k = 2$, let C_2^v be a circuit computing f_2^v; that is, C_2^v is an OR of v AND's, each having fanin $= \sqrt{v}$. Let C be a $\Delta_2(r)$-circuit, with $r < \delta v^{1/5}$. We will show that C does not compute f_2^v; that is, we will find a restriction ρ such that $C_2^v\lceil_\rho \neq C\lceil_\rho$.

Consider C, a $\Delta_2(r)$-circuit. Recall that, for each AND gate g_l at second level of C, g_l is labeled with r bits, $b_{l_1}b_{l_2}\cdots b_{l_r}$, and if $b_{l_i} = 1$, then the ith subcircuit from the left of g_l is a $\Sigma_1(r)$-circuit, and if $b_{l_i} = 0$, then the ith subcircuit from the left of g_l is a $\Pi_1(r)$-circuit. Also, for any other AND gate $g_{l'}$ at the second level with label $b_{l'_1}b_{l'_2}\ldots b_{l'_r}$, where $b_{l_j} = b_{l'_j}$ for $1 \leq j < i$ and $b_{l_i} \neq b_{l'_i}$, the ith subcircuits of g_l and g'_l are dual circuits. Thus, if we find a restriction ρ that makes the ith subcircuit of g_l output 1, then the ith subcircuit of $g_{l'}$ will output 0 by Proposition 2.1. Therefore, the entire subcircuit g'_l at level 2 of C can be replaced by a constant 0. By making assignments to at most r^2 variables, we will be able to determine completely the output of C while the output of C_2^v remains undetermined.

The process of finding an appropriate assignment of variables starts by fixing the leftmost subcircuit of each second-level AND gate of C. Note that there is a single $\Sigma_1(r)$ circuit C_1 such that the leftmost subcircuit of each second-level AND gate of C is one of C_1 or $\hat{C}_1$. If C_1 can be forced to output 1 by assigning at most r variables (recall that C_1 is an OR of ANDs, each AND with fanin at most r), then this is done. Then, all second level AND gates of C with C_1 as the leftmost subcircuit are modified by replacing C_1 with 1, and all second level AND gates of C with $\hat{C}_1$ as the leftmost subcircuit are removed from C. On the other hand, if C_1 cannot be forced to output 1, then $\hat{C}_1$ will always output 1 regardless of assignments to variables. In this case, all second-level AND gates of C with $\hat{C}_1$ as the leftmost subcircuit are modified by replacing $\hat{C}_1$ with 1, all second-level AND gates of C with C_1 as the leftmost subcircuit are removed from C, and no assignments are made. In either case, all second-level AND gates that now remain in C had the same leftmost subcircuit, either C_1 or $\hat{C}_1$, which is now replaced by 1. This implies that there is a single $\Sigma_1(r)$ circuit C_2 such that the second subcircuit of each remaining second-level AND gate of C is one of C_2 or $\hat{C}_2$. Hence, we can repeat the same process with C_2, then with some C_3, etc., until some C_r. At this point, at most r^2 variables will have been assigned and the output of C is completely determined. Since $r^2 \leq (\delta v^{1/5})^2 < \sqrt{v}$ for sufficiently large v, there is no bottom AND gate G of C_2^v such that $G\lceil_\rho = 1$. Also, since the top fanin of C_2^v is v, there is some bottom AND gate G' of C_2^v such that no variables of G' have been decided in the previous process; that is, we can find a restriction ρ such that

$$(\#) \qquad C\lceil_\rho \neq * \quad \text{and} \quad C_2^v\lceil_\rho = *.$$

Thus, C cannot compute a function f_2^v when the size of C is less than $\delta v^{1/5}$. This establishes the base case for the inductive proof of Theorem 3.3.

Our induction hypothesis is that for some fixed $k' \geq 2$ and for sufficiently large v, $s_{k'}(v) \geq \delta v^{1/5}$; that is, for sufficiently large v, if C is a $\Delta_{k'}(r)$-circuit with $r < \delta v^{1/5}$, then C does not compute the function $f_{k'}^v$. For the induction step, consider $k = k' + 1 > 2$. Consider a C_k^v-circuit that computes an f_k^v function. Let $q = v^{-1/3}$ and let $\mathcal{B} = \{B_j\}$, where B_j is the set of variables of the bottom AND gate H_j of C_k^v. By way of contradiction, assume that there exists a circuit C, that is, a $\Delta_k(r)$-circuit with $r < \delta v^{1/5}$, computing the function f_k^v. We will argue that after applying a random restriction ρ from $R_{q,\mathcal{B}}^+$, if k is even, or from $R_{q,\mathcal{B}}^-$, if k is odd, to C, the resulting circuit $C\lceil_{\rho g(\rho)}$ is a $\Delta_{k-1}(r)$-circuit with $r < \delta v^{1/5}$ that computes the function f_{k-1}^v, which contradicts our induction hypothesis.

To apply the induction hypothesis, we need to verify that

(1) with a high probability, $C\lceil_{\rho g(\rho)}$ is equivalent to a $\Delta_{k-1}(r)$-circuit,

and

(2) with a high probability, $C_k^v\lceil_{\rho g(\rho)}$ contains a subcircuit computing the function f_{k-1}^v.

Part (1) is proved by Lemma 2.8. To apply Lemma 2.8, let us choose $\gamma = \delta v^{1/5}$. We have $\alpha < 6q\gamma = 1/8 \cdot v^{-2/15}$. By definition, the total number of $\Sigma_k(r)$-circuits and $\Pi_k(r)$-circuits in a $\Delta_k(r)$-circuit is bounded by $r \cdot 2^r \leq 2^{2r}$. Thus, we only need to show that $2^{2r} < 2^{2\gamma} \leq 1/(3(2\alpha)^\gamma)$. We have

$$2^{2\gamma} \cdot 3(2\alpha)^\gamma = 3 \cdot (8\alpha)^\gamma \leq 3 \cdot (v^{-2/15})^\gamma \leq 1$$

for sufficiently large v. This proves part (1). Part (2) was proved in Lemma 2.9. This completes the proof of Theorem 3.3. □

To satisfy requirement R_n', notice that C_n is a $\Delta_k(q(m))$-circuit corresponding to the predicate $0^m \in L(M_n, K^{k-1}(A))$ and that C_0, corresponding to the predicate $0^m \in L_k^A$, contains a subcircuit that computes an $f_k^{2^{m/(k+2)}}$ function. Thus, we need to choose an integer

m so large such that $q(m) < (1/48) \cdot 2^{m/5(k+2)}$ and that Theorem 3.3 holds for $s_k(2^{m/(k+2)})$. We now have the desired theorem.

THEOREM 3.4. *There exists an oracle A such that for all $k \geq 1$, $\Delta_k^P(A) \subsetneq \Sigma_k^P(A)$.*

3.2. Separating Θ-classes and Δ-classes in relativized polynomial-time hierarchies. In this section, we show that there exists a recursive oracle A such that for all $k \geq 2$, $\Theta_k^P(A) \neq \Delta_k^P(A)$. We will find, for each $k \geq 2$, a set $L_{\text{odd},k}^A \in \Delta_k^P(A)$ and that $L_{\text{odd},k}^A \notin \Theta_k^P(A)$. Again, by dovetailing the diagonalization for each $k \geq 2$, the oracle set A can be constructed so that $\Theta_k^P(A) \subsetneq \Delta_k^P(A)$ for all $k \geq 2$.

We will again use circuit lower bound techniques to prove this separation. Similarly to finding special circuits that represent sets in $\Delta_k^P(A)$, we need special circuits that represent sets in $\Theta_k^P(A)$. Consider a deterministic polynomial-time bounded oracle machine M using oracle set B and, on any input x, the query tree T that represents all the possible computations of M on x. Since there are at most $O(\log |x|)$ queries in any one computation path, the total number of possible paths is bounded by a polynomial $p(|x|)$. Thus, we can define $\Theta_k(m)$-circuits in a manner that is similar to the definition of $\Delta_k(m)$-circuits.

DEFINITION 3.5. *For $k \geq 2$, C is a $\Theta_k(m)$-circuit if and only if:*

(1) *C has $k+2$ levels, and the top gate is an* OR *with fanin $\leq m$;*

(2) *the gates on second level are* AND *gates with fanins $\leq \log m$;*

(3) *each* AND *gate G_j at the second level has distinct label of $n = \log m$ bits, $b_1 \ldots b_n$, and if $b_i = 1$ then, the ith subcircuit from the left of G_j is a $\Sigma_{k-1}(m)$-circuit, and if $b_i = 0$, then the ith subcircuit from the left of G_j is a $\Pi_{k-1}(m)$-circuit;*

(4) *for any two* AND *gates G_j and G_n, labeled l_j and l_n, at the second level, if the first r bits, $r < \log m$, of labels l_j and l_n are the same, then the first r subcircuits from the left of G_j and G_n are the same; and*

(5) *for any two* AND *gates G_j and G_n, labeled l_j and l_n, at the second level, if the first bit from the left of l_j and l_n where l_j and l_n differ is b_r, $r \leq \log m$, then the rth subcircuits from the left of G_j and G_n are dual circuits.*

LEMMA 3.6. *Let $k \geq 2$. For every set $L \in \Theta_k^P(A)$, there is a polynomial q such that for every input string x, there is a corresponding $\Theta_k(q(|x|))$-circuit $C_{M,x}$ such that $x \in L \Leftrightarrow C_{M,x}\lceil_{\rho_A} = 1$.*

To separate $\Theta_k^P(A)$ and $\Delta_k^P(A)$, for all $k \geq 2$, we define

$$L_{\text{odd},k}^A = \{x0^{kn} \mid |x| = n, \text{ and } \max\{y \mid |y| = |x|, xy\,0^{(k-1)n} \in L_{k-1}^A\} \text{ has odd parity}\}.$$

Recall that L_{k-1}^A, as defined in §3.1, is a set in $\Sigma_{k-1}^P(A)$; thus $L_{\text{odd},k}^A$ is in $\Delta_k^P(A)$ using a binary search algorithm. The setup of the diagonalization is similar to the one in §3.1 with functions $t(n)$ and $D(n)$. For each stage $n > 0$ of the construction, we want to satisfy the following requirement R_n:

$$R_n : (\exists x = 0^m)(\exists B \subseteq D(n))[0^m \in L_{\text{odd},k}^B \Leftrightarrow 0^m \notin L(M_n, K^{k-1}(A'))],$$

where $A' = A(n-1) \cup B$ and $A(n-1)$ denotes the strings in A decided prior to stage n, and M_n is the nth Θ_k^P oracle machine making at most $\log(p_n(m))$. Note that the predicate "$0^m \in L_{\text{odd},k}^B$" can be represented by a k-level circuit with a top MaxOdd gate having fanin $2^{m/(k+1)}$. Each top level subcircuit, C_y, of MaxOdd contains a $C_{k-1}^{2^{m/(k+1)}}$-subcircuit computing an $f_{k-1}^{2^{m/(k+1)}}$ function, where y corresponds to the y in the definition of $L_{\text{odd},k}^B$. Also, the subcircuits C_y will be arranged, in lexicographic order on y, from left to right as the children of the top MaxOdd gate. Let C_0 be the corresponding k-level circuit for the predicate $0^m \in L_{\text{odd},k}^B$. C_0 contains an $H_k^{2^{m/(k+1)}}$ subcircuit that computes an $h_k^{2^{m/(k+1)}}$ function.

By Lemma 3.6, there exist a polynomial q and a $\Theta_k(q(m))$-circuit C_n corresponding to the the machine M_n on input 0^m, with oracle $K^{k-1}(A)$. We let $t(n) = q(m)$. The requirement

R_n can then be rewritten as

$$R'_n : (\exists x = 0^m)(\exists B \subseteq D(n))[C_0\lceil_{\rho_B}= 1 \Leftrightarrow C_n\lceil_{\rho_{A'}}= 0],$$

where $A' = A(n-1) \cup B$.

Now let $s_k(v)$ be the minimum r for a $\Theta_k(r)$-circuit to compute an h_k^v function. The following lower bound on $s_k(v)$ shows that the requirement R'_n can be satisfied.

THEOREM 3.7. *For all $k \geq 2$, for sufficiently large v, $s_k(v) \geq \delta v^{1/5}$, where $\delta = 1/24$.*

Proof. Again, the proof is by induction on k. For the base case of $k = 2$, let H_2^v be an MaxOdd of v subcircuits, each computing a function f_1^v. Let C be a $\Theta_2(r)$-circuit, with $r < \delta v^{1/5}$. We need to show that C does not compute the same function as H_2^v; that is, we will find a restriction ρ such that $H_2^v\lceil_\rho \neq C\lceil_\rho$.

Consider C, the $\Theta_2(r)$-circuit. Accounting for the fact that some Σ_1- and Π_1-circuits are duals, there are at most $r < \delta v^{1/5}$ many different Σ_1-circuits and Π_1-circuits in C. (It is easy to see this by looking at the query tree of a Θ_2^P oracle machine.) During the construction of A at stage n, we will put strings into A step by step. At each step we place some strings into A in order to flip-flop the output of $H_2^v\lceil_{\rho_A}$. During the same step, at least one of the Σ_1-circuits (or Π_1-circuits) of C will be fixed to have output 1 (or 0) by assigning variables to A only. Since there are only $r < \delta v^{1/5}$ many different Σ_1-circuits and Π_1-circuits, we will eventually fix the output of C after at most $r + 1 \leq \delta v^{1/5}$ flip-flops of H_2^v. We argue that, after this process, we are still able to flip-flop the output of H_2^v as we desire. Thus, we can find the needed restriction. Now we present this flip-flop process.

Process FF:

```
$B :=$ the variables of the leftmost OR gate of $H_2^v$; $\overline{B} := \emptyset$;
while $C\lceil_{\rho_B}= H_2^v\lceil_{\rho_B}$ do
    if there is some $\Sigma_1$-circuit $C_j$ of $C$ with $C_j\lceil_{\rho_B}= 1$ then
        for all $\Sigma_1$-circuits $C_j$ of $C$ with $C_j\lceil_{\rho_B}= 1$ do
            Fix the output of one AND gate $G$ of $C_j$ to be 1 with restriction $\rho_B$
            by assigning all negative variables of $G$ to $\overline{B}$
        end(* for *)
    end(* if *)
    if there is some $\Pi_1$-circuit $C'_j$ of $C$ with $C'_j\lceil_{\rho_B}= 0$ then
        for all $\Pi_1$-circuits $C'_j$ of $C$ with $C'_j\lceil_{\rho_B}= 0$ do
            Fix the output of one OR gate $G'$ of $C'_j$ to be 0 with restriction $\rho_B$
            by assigning all positive variables of $G'$ to $\overline{B}$
        end(* for *)
    end(* if *)
    Find the leftmost subcircuit $H_i$ of $H_2^v$ such that $H_i\lceil_{\rho_B}= 0$.
    Let $V$ be the set of variables in $H_i$ that are not in $\overline{B}$. Let $B := B \cup V$.
end(* while *)
```

We now carefully analyze **Process FF**. Initially, B contains the variables in the leftmost OR gate of H_2^v and $\overline{B}$ is empty. We have $H_2^v\lceil_{\rho_B}= 0$. If $H_2^v\lceil_{\rho_B}\neq C\lceil_{\rho_B}$, then we are done. Otherwise, we will try to fix the output of some of the Σ_1-circuits or Π_1-circuits of C. To make sure that we only reserve polynomial many strings in the entire process, we only consider fixing the output of Σ_1-circuits to be 1 and the output of Π_1-circuits to be 0. This process is done in the following steps: for all the Σ_1-circuits C_j of C such that $C_j\lceil_{\rho_B}= 1$, fix the output of one of C_j's bottom AND gates G to be 1 with restriction ρ_B. If we assign all the negative variables in G to $\overline{B}$, then we will preserve the output of C_j in later steps. Note that the positive variables of G are already in B since G is an AND gate and $G\lceil_{\rho_B}$ outputs 1. In later steps, we will never reassign the variables in B or $\overline{B}$. Also, for each Σ_1-circuit, there are at most r

many variables that are assigned to $\overline{B}$ in this process. We carry out a similar process for the Π_1-circuits C'_j of C to obtain $C'_j\lceil_{\rho_B}= 0$.

The next step in **Process FF** is to "flip-flop" the output of H_2^v. This is done by adding the free variables of the leftmost subcircuit H_i of H_2^v such that $H_i\lceil_{\rho_B}= 0$. We will assure that there are always some free variables in H_i that are not in $\overline{B}$. This will flip-flop H_i's output between 0 and 1, and the output of the other OR gates in H_2^v remains unchanged (recall that all variables in H_2^v are unique). As output of H_2^v is flip-flopped, if the output of $C\lceil_{\rho_B}$ does not flip-flop, we are done. *Otherwise, if $C\lceil_{\rho_B}$ also flip-flops its output, there must be some Σ_1-circuits (Π_1-circuits) that output* 0 (1) *in the previous iteration but then change their outputs to* 1 (0). This is because Σ_1-circuits outputting 1 and Π_1-circuits outputting 0 are already fixed. In this case, we repeat the same process. Note that at each iteration of the process, at least one Σ_1- or Π_1-circuit is fixed.

Now we argue that the variables reserved in $\overline{B}$ will not make any H_i's of H_2^v permanently output 0; that is, flip-flopping the output of H_2^v is always possible. Observe the following two facts.

(1) There are at most $r < \delta v^{1/5}$ Σ_1-circuits and Π_1-circuits in C. Thus, after at most $r + 1$ iterations, the while-loop in **Process FF** terminates.

(2) There are v subcircuits of the top MaxOdd gate of H_2^v and each subcircuit is an OR of $\sqrt{v}$ inputs.

Since we assign at most r variables to $\overline{B}$ for each Σ_1-circuit and Π_1-circuit, there are, by fact (1), at most $r(r + 1) \leq (\delta v^{1/5})^2 < \sqrt{v}$ variables reserved to $\overline{B}$ during **Process FF**. Thus, none of the H_i's of H_2^v will permanently output 0 based on assignments to $\overline{B}$. Finally, since there are at most $r + 1 \leq v^{1/5}$ iterations of the while-loop and since H_2^v can flip-flop at least v times, we then conclude that there is a pair of finite sets B and $\overline{B}$ that

$$(*) \qquad C\lceil_{\rho_{B,\overline{B}}}\neq * \quad \text{and} \quad H_2^v\lceil_{\rho_{B,\overline{B}}}= *.$$

This completes the base case of $k = 2$.

Our induction hypothesis is that for some fixed $k' \geq 2$ and for sufficiently large v, $s_{k'}(v) \geq \delta v^{1/5}$; that is, for sufficiently large v, if C is a $\Theta_{k'}(r)$-circuit with $r < \delta v^{1/5}$, then C does not compute the function $h_{k'}^v$. For the induction step, consider $k = k' + 1 > 2$. Consider an H_k^v-circuit that computes an h_k^v function. Let $q = v^{-1/3}$ and let $\mathcal{B} = \{B_j\}$, where B_j is the set of variables of the bottom AND gate G_j of H_k^v. By way of contradiction, assume that there exists a circuit C, that is, a $\Theta_k(r)$-circuit with $r < \delta v^{1/5}$, computing the function h_k^v. We will argue that after applying a random restriction from $R_{q,\mathcal{B}}^+$, if k is odd or from $R_{q,\mathcal{B}}^-$, if k is even, to C, the resulting circuit $C\lceil_{\rho g(\rho)}$ is a $\Theta_{k-1}(r)$-circuit with $r < \delta v^{1/5}$ that computes the function h_{k-1}^v, which contradicts our induction hypothesis.

To apply the induction hypothesis, we need to verify that

(1) with a high probability, $C\lceil_{\rho g(\rho)}$ is equivalent to a $\Theta_{k-1}(r)$-circuit,

and

(2) with a high probability, $H_k^v\lceil_{\rho g(\rho)}$ contains a subcircuit computing the function h_{k-1}^v.

Again, part (1) is proved by the Lemma 2.8. To apply Lemma 2.8, we choose $\gamma = \delta v^{1/5}$. We have $\alpha < 6q\gamma = 1/4 \cdot v^{-2/15}$. The total number of $\Sigma_{k-1}(r)$-circuits and $\Pi_{k-1}(r)$-circuits of C is bounded by $r \cdot \log r \leq 2^r < 2^\gamma$. To apply Lemma 2.8, we need to show that $2^\gamma \leq 1/(3(2\alpha)^\gamma)$. We have

$$2^\gamma \cdot 3(2\alpha)^\gamma = 3(4\alpha)^\gamma \leq 3(v^{-2/15})^\gamma \leq 1,$$

for sufficiently large v. This proves part (1). Part (2) follows from Lemma 2.9. This completes the proof of Theorem 3.7. □

To satisfy requirement R'_n, we need to choose an integer m so large that Theorem 3.7 holds for $s_k(2^{m/(k+1)})$ and such that $q(m) < (1/24) \cdot 2^{m/5(k+1)}$.

THEOREM 3.8. *There exists an oracle A such that for all $k \geq 2$, $\Theta_k^P(A) \subsetneq \Delta_k^P(A)$.*

3.3. Separating Σ-classes and Θ-classes in relativized polynomial-time hierarchies. In this section, we show that there exists a recursive oracle A such that for all $k \geq 2$, $\Sigma_{k-1}^P(A) \neq \Theta_k^P(A)$. We will find, for each $k \geq 2$, a set $L_{\mathrm{xor},k}^A \in \Theta_k^P(A)$ such that $L_{\mathrm{xor},k}^A \notin \Sigma_{k-1}^P(A)$. Once again, by dovetailing the diagonalization for each $k \geq 2$, the oracle set A can be constructed so that $\Sigma_{k-1}^P(A) \subsetneq \Theta_k^P(A)$ for all $k \geq 2$.

To separate $\Sigma_{k-1}^P(A)$ and $\Theta_k^P(A)$, for all $k \geq 2$, we define

$$L_{\mathrm{xor},k}^A = \{x0^{kn} \mid |x| = n, \text{ and exactly one of } 0^n x 0^{(k-1)n} \text{ or } 1^n x 0^{(k-1)n} \text{ is in } L_{k-1}^A\},$$

where L_{k-1}^A is a set in $\Sigma_{k-1}^P(A)$, as defined in §3.1. Thus $L_{\mathrm{xor},k}^A$ is in $\Theta_k^P(A)$.

The setup of the diagonalization is similar to the one in §3.1. For each stage $n > 0$, we want to satisfy the following requirement R_n:

$$R_n : (\exists x = 0^m)(\exists B \subseteq D(n))[0^m \in L_{\mathrm{xor},k}^B \Leftrightarrow \sigma_n(A; 0^m) \text{ is false }],$$

where σ_n is the nth $\Sigma_{k-1}^{P,1}$-predicate. Note that the predicate "$0^m \in L_{xor,k}^B$?" can be represented by a k-level circuit with a top XOR gate having fanin of 2. Each subcircuit of the XOR gate contains a $C_{k-1}^{2^{m/(k+1)}}$-subcircuit computing an $f_{k-1}^{2^{m/(k+1)}}$ function. Let C_0 be the corresponding k-level circuit for the predicate $0^m \in L_{\mathrm{xor},k}^B$. C_0 contains a $D_k^{2^{m/(k+1)}}$-subcircuit that computes an $l_k^{2^{m/(k+1)}}$ function.

By Lemma 2.3, there exist a polynomial q and a $\Sigma_{k-1}(q(m))$-circuit C_n corresponding the predicate $\sigma_n(A; 0^m)$. Again, let $t(n) = q(m)$. The requirement R_n can then be rewritten as

$$R_n' : (\exists x = 0^m)(\exists B \subseteq D(n))[C_0\lceil_{\rho_B} = 1 \Leftrightarrow C_n\lceil_{\rho_{A'}} = 0],$$

where $A' = A(n-1) \cup B$ and $A(n-1)$ is the set of strings in A decided prior to stage n. Now let $s_k(v)$ be the minimum r such that a $\Sigma_{k-1}(r)$-circuit can compute an l_k^v function. The following lower bound on $s_k(v)$ shows that the requirement R_n' can be satisfied.

THEOREM 3.9. *For all $k \geq 2$, for sufficiently large v, $s_k(v) \geq \delta v^{1/3}$, where $\delta = 1/12$.*

Proof. Again, the proof is by induction on k. For the base case of $k = 2$, let D_2^v be an XOR of 2 subcircuits D_1 and D_2, each computing an f_1^v function. Let C be a $\Sigma_1(r)$-circuit, where $r < \delta v^{1/3}$. We will find a restriction ρ such that $C\lceil_\rho \neq *$ and $D_2^v\lceil_\rho = *$.

Let v_1 be the leftmost variable of D_1. Initially, let $B = \overline{B} = \emptyset$. Let $\rho_\emptyset$ denote the restriction that assigns 0 to all the variables, let $\rho_{\{v_1\}}$ denote the restriction that assigns 1 to v_1 and 0 to all the other variables, and let $\rho_{\emptyset, \Sigma^* - \{v_1\}}$ denote the restriction that assigns $*$ to v_1 and assigns 0 to all the other variables. There are three cases:

Case 1. $C\lceil_{\rho_\emptyset} = 1$. In this case, there must be an AND gate G of C with only negative variables. Assign all the variables in G to $\overline{B}$. The number of variables being assigned is bounded by $r < \delta v^{1/3} < \sqrt{v}$. Thus, neither D_1 nor D_2 will be forced to output a constant by $\rho_{B,\overline{B}}$ and $D_2^v\lceil_{\rho_{B,\overline{B}}} = *$, where $B = \emptyset$.

Case 2. $C\lceil_{\rho_\emptyset} = 0$ and $C\lceil_{\rho_{\{v_1\}}} = 0$. In this case, changing the assignment of v_1 from 0 to 1 does not change the output of $C\lceil_{\rho_\emptyset}$. Thus, $C\lceil_{\rho_{\emptyset,\Sigma^*-\{v_1\}}} = 0$. $D_2^v\lceil_{\rho_{\emptyset,\Sigma^*-\{v_1\}}} = *$ since the output of $D_2^v\lceil_{\rho,\Sigma^*}$ can be set to 1 by assigning 1 to variable v_1 or left at 0 by assigning 0 to v_1.

Case 3. $C\lceil_{\rho_\emptyset} = 0$ and $C\lceil_{\rho_{\{v_1\}}} = 1$. In this case, there is an AND gate G of C that outputs 1 by $\rho_{\{v_1\}}$. Since v_1 is the only variable that is assinged 1 by $\rho_{\underline{\{v_1\}}}$, all the other variables in G must be negative variables. Assign all negative variables to $\overline{B}$ and assign v_1 to B. Since the number of negative variables is bounded by $r < \delta v^{1/3} < \sqrt{v}$ and since only one positive variable has been assigned, it is easy to see that $C\lceil_{\rho_{B,\overline{B}}} = 1$ and $D_2^v\lceil_{\rho_{B,\overline{B}}} = *$.

Our induction hypothesis is that for some fixed $k' \geq 2$ and for sufficiently large v, $s_{k'}(v) \geq \delta v^{1/3}$; that is, for sufficiently large v, if C is a $\Sigma_{k'-1}(r)$-circuit with $r < \delta v^{1/3}$, then C does not comupte the function $l^v_{k'}$. For the induction step, consider $k = k' + 1 > 2$. Consider a D^v_k-circuit that computes an l^v_k function. Let $q = v^{-1/3}$ and let $\mathcal{B} = \{B_j\}$, where B_j is the set of variables of the bottom AND gate G_j of D^v_k. By way of contradiction, assume that there exists a circuit C, that is, a $\Sigma_{k-1}(r)$-circuit with $r < \delta v^{1/3}$, computing the function l^v_k. We will argue that after applying a random restriction from $R^+_{q,\mathcal{B}}$, if k is odd, or from $R^-_{q,\mathcal{B}}$, if k is even, to C, the resulting circuit $C\lceil_{\rho g(\rho)}$ is a $\Sigma_{k-2}(r)$-circuit with $r < \delta v^{1/3}$ that computes the function l^v_{k-1}, which contradicts our induction hypothesis.

For induction step, let C be a $\Sigma_{k-1}(r)$-circuit and let D^v_k be the circuit computing an l^v_k function. To apply the induction hypothesis, we need to verify that

(1) with a high probability, $C\lceil_{\rho_{\rho,g(\rho)}}$ is equivalent to a $\Sigma_{k-2}(r)$-circuit, and

(2) with a high probability, $D^v_k\lceil_{\rho g(\rho)}$ contains a subcircuit computing the function l^v_{k-1}.

Part (1) is proved by Lemma 2.7. To apply Lemma 2.7, we choose $\gamma = \delta v^{1/3}$. We have $\alpha < 6q\gamma = 1/2$. The total number of depth-2 subcircuits at the bottom level of C is bounded by its size $2^r < 2^\gamma$. We need to show that $2^\gamma \leq 1/(3 \cdot \alpha^\gamma)$. We have

$$2^\gamma \cdot 3 \cdot \alpha^\gamma = 3(2\alpha)^{\delta v^{1/3}} \leq 1,$$

for sufficiently large v. Part (2) follows from Lemma 2.9. This completes the proof of Theorem 3.9. □

To satisfy requirement R'_n, we need to choose an integer m so large that Theorem 3.9 holds for $s_k(2^{m/(k+1)})$ and such that $q(m) < (1/12) \cdot 2^{m/3(k+1)}$.

THEOREM 3.10. *There exists an oracle A such that for all $k \geq 2$, $\Sigma^P_{k-1}(A) \subsetneq \Theta^P_k(A)$.*

The constructions of Theorems 3.3, 3.7, and 3.9 can all be dovetailed to obtain the following:

THEOREM 3.11. *For all $k \geq 2$, there exists an oracle A such that $\Sigma^P_{k-1}(A) \subsetneq \Theta^P_k(A) \subsetneq \Delta^P_k(A) \subsetneq \Sigma^P_k(A)$.*

4. The extended low hierarchy has an infinite number of distinct levels. An interesting question is whether the low hierarchy or the extended low hierarchy consists of an infinite number of distinct levels. The existence of an infinite number of distinct levels in the low hierarchy not only implies that P $\neq$ NP but also provides a natural complexity categorization of sets in NP: sets at the higher levels of the low hierarchy are more "complicated" than sets at the lower levels. It is interesting to note that there exists an oracle set relative to which the low hierarchy is a properly infinite hierarchy and that there also exists an oracle set relative to which the low hierarchy has a finite number of distinct levels [Ko91].

The situation is different for the extended low hierarchy. In proving optimal lower bounds for the extended low hierarchy, Allender and Hemachandra [AH92] constructed a sparse set that is not in $EL^{P,\Sigma}_2$, level 2 of the extended low hierarchy based on the Σ-levels of the polynomial-time hierarchy. It is known that sparse sets are in $EL^{P,\Theta}_3$, level 3 of the extended low hierarchy based on the Θ-levels of the polynomial-time hierarchy. This shows that the extended low hierarchy has at least two distinct levels.

In this section, we show that the extended low hierarchy actually consists of an infinite number of distinct levels. Our theorems are motivated by the observation that there are hard sets in the extended low hierarchy. As noted in [BBS86], any PSPACE complete set, for example, is in the extended low hierarchy. Sets, like hard sets, that can collapse the polynomial-time hierarchy to a fixed level, but no lower, are candidates for sets separating various levels of the extended low hierarchy.

To better understand why this happens, first consider the definition of the extended low hierarchy.

DEFINITION 4.1 [BBS86], [AH92], [LS91].(1) *For each* $k \geq 1$, $EL_k^{P,\Sigma} = \{A \mid \Sigma_k^P(A) \subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT})\}$.

(2) *For each* $k \geq 2$, $EL_k^{P,\Delta} = \{A \mid \Delta_k^P(A) \subseteq \Delta_{k-1}^P(A \oplus \mathrm{SAT})\}$.

(3) *For each* $k \geq 2$, $EL_k^{P,\Theta} = \{A \mid \Theta_k^P(A) \subseteq \Theta_{k-1}^P(A \oplus \mathrm{SAT})\}$.

Proposition 4.2 shows the basic relationships between the levels of the extended low hierarchy.

PROPOSITION 4.2 [AH92], [LS91]. (1) *For each* $k \geq 1$, $EL_k^{P,\Sigma} \subseteq EL_{k+1}^{P,\Theta} \subseteq EL_{k+1}^{P,\Delta} \subseteq EL_{k+1}^{P,\Sigma}$.

(2) $EL_1^{P,\Sigma} = EL_2^{P,\Theta} = EL_2^{P,\Delta}$.

The following observation is very useful: for all $k \geq 1$ and for all sets A,

(1) $\Sigma_k^P(A) \subseteq \Sigma_k^P(A \oplus \mathrm{SAT}) \subseteq \Sigma_{k+1}^P(A)$,

(2) $\Delta_k^P(A) \subseteq \Delta_k^P(A \oplus \mathrm{SAT}) \subseteq \Delta_{k+1}^P(A)$,

(3) $\Theta_{k+1}^P(A) \subseteq \Theta_{k+1}^P(A \oplus \mathrm{SAT}) \subseteq \Theta_{k+2}^P(A)$.

Thus, if a set A is so powerful that $\Sigma_{k+1}^P(A)$ collapses to $\Sigma_k^P(A)$, then A, by definition, is in the extended low hierarchy. This also holds for Δ-levels and Θ-levels of the polynomial-time hierarchy as well. We express this property in the following proposition.

PROPOSITION 4.3. (1) $\forall k \geq 0, \forall A\ [\Sigma_{k+1}^P(A) \subseteq \Sigma_k^P(A) \Rightarrow A \in EL_{k+1}^{P,\Sigma}]$.

(2) $\forall k \geq 1, \forall A\ [\Delta_{k+1}^P(A) \subseteq \Delta_k^P(A) \Rightarrow A \in EL_{k+1}^{P,\Delta}]$.

(3) $\forall k \geq 1, \forall A\ [\Theta_{k+1}^P(A) \subseteq \Theta_k^P(A) \Rightarrow A \in EL_{k+1}^{P,\Theta}]$.

The proofs of our main theorems, separating the levels of the extended low hierarchy, are motivated by Ko's recent result that, for each $k \geq 1$, there exists an oracle set relative to which the polynomial-time hierarchy has exactly k levels [Ko89]:

$$\forall k \geq 1, \exists A\ [\mathrm{PH}(A) = \Sigma_k^P(A) \supsetneq \Sigma_{k-1}^P(A)].$$

Note that this result strongly suggests that the Σ-levels of the extended low hierarchy are all distinct. Specifically, if $\mathrm{PH}(A) = \Sigma_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT})$, then $A \in EL_{k+1}^{P,\Sigma} - EL_k^{P,\Sigma}$. It is a simple matter to modify Ko's result to show that the Σ-levels of the extended low hierarchy are all distinct. We significantly extend Ko's work by showing that each of the levels of the extended low hierarchy are distinct; that is,

$$\forall k \geq 2, EL_k^{P,\Sigma} \subsetneq EL_{k+1}^{P,\Theta} \subsetneq EL_{k+1}^{P,\Delta} \subsetneq EL_{k+1}^{P,\Sigma}.$$

4.1. Separating $EL_k^{P,\Sigma}$ and $EL_k^{P,\Delta}$. In this section, we will separate $EL_k^{P,\Sigma}$ and $EL_k^{P,\Delta}$ for each $k \geq 2$; that is, for arbitrary $k \geq 2$, we will construct a set A such that $A \in EL_k^{P,\Sigma}$ and $A \notin EL_k^{P,\Delta}$. From Proposition 4.3, we know that if $\Sigma_k^P(A) \subseteq \Sigma_{k-1}^P(A)$, then A is in $EL_k^{P,\Sigma}$. To separate $EL_k^{P,\Sigma}$ and $EL_k^{P,\Delta}$, we need to show that $\Delta_k^P(A) \not\subseteq \Delta_{k-1}^P(A \oplus \mathrm{SAT})$. Since $\Sigma_{k-1}^P(A) \subseteq \Delta_k^P(A)$, we will construct a set A such that $\mathrm{PH}(A)$ collapses to $\Sigma_{k-1}^P(A)$ and $\Sigma_{k-1}^P(A) \not\subseteq \Delta_{k-1}^P(A \oplus \mathrm{SAT})$. As a result, $\Delta_k^P(A) \not\subseteq \Delta_{k-1}^P(A \oplus \mathrm{SAT})$. For the case of $k = 2$, it is known that there exists an oracle A such that $\mathrm{PH}(A) = \mathrm{NP}(A) \neq \mathrm{P}(A)$ [BGS75]. A direct modification yields that $\mathrm{PH}(A) = \mathrm{NP}(A) \not\subseteq \mathrm{P}(A \oplus \mathrm{SAT})$. The case of $k \geq 3$ is handled in the following theorem.

THEOREM 4.4. *For every recursive set B, and for each $k \geq 2$, there exists a recursive set A such that* $\mathrm{PH}(A) = \Sigma_k^P(A) \not\subseteq \Delta_k^P(A \oplus B)$.

Proof. Our proof will again use circuit lower bound techniques. For arbitrary $k \geq 2$, we need to collapse $\mathrm{PH}(A)$ to $\Sigma_k^P(A)$ and at the same time separate $\Sigma_k^P(A)$ and $\Delta_k^P(A \oplus B)$. To collapse PH to $\Sigma_k^P(A)$, we will encode the $\Sigma_{k+1}^P(A)$ complete set $K^{k+1}(A)$ into a set in $\Sigma_k^P(A)$. To separate $\Delta_k^P(A \oplus B)$ and $\Sigma_k^P(A)$, we will use the set L_k^A defined in the previous section:

$$L_k^A = \{x0^{kn} \mid |x| = 2n, (\exists y_1, |y_1| = n)(\forall y_2, |y_2| = n) \cdots (Q_k y_k, |y_k| = n) 0xy_1y_2 \ldots y_k \in A\}.$$

For each $k \geq 1$, define the predicate

$$\tau_k(A; x) \equiv (\exists y_1, |y_1| = n) \cdots (Q_k y_k, |y_k| = n)\ 1xy_1y_2 \ldots y_k \in A,$$

where $|x| = 2n$. Thus, for each x of length $2n$, the value of $\tau_k(A; x)$ only depends on strings in A of length $(k+2)n+1$. Note that $\tau_k(A; x)$ is false if $|x|$ is odd. It is easy to verify that $\tau_k(A; x)$ is a $\Sigma_k^{P,1}$-predicate. Also, it is clear that, for any pair of strings x and y, whether $x \in L_k^A$ and $\tau_k(A; y)$ are true depends on the membership in A of different strings.

The construction of A will make sure that for all strings $x \in \Sigma^*$, $x \in K^{k+1}(A) \Leftrightarrow \tau_k(A; x0^{|x|})$. Thus, $\mathrm{PH}(A) = \Sigma_{k+1}^P(A) = \Sigma_k^P(A)$. We will also construct A so that $L_k^A \notin \Delta_k^P(A \oplus B)$. More specifically, A will be constructed by stages. Let $M_1, M_2, \ldots$ be an effective enumeration of deterministic polynomial time-bounded oracle machines. The setup of $t(n)$ and $D(n)$ is similar to that in the previous section. Let m be a multiple of $k+2$. At stage n of the construction, we will find an assignment of strings to A and $\overline{A}$ so that $0^m \in L_k^A$ if and only if $0^m \notin L(M_n, K^{k-1}(A \oplus B))$. In stage n, we also "$\Sigma_k^P(A)$-encode" into A strings $x \in K^{k+1}(A)$, $t(n-1)+1 \leq (k+2)\cdot|x|+1 \leq t(n)$, such that $x \in K^{k+1}(A) \Leftrightarrow \tau_k(A; x0^{|x|})$. The encoding is divided into two steps: before and after finding the witness 0^m separating $\Sigma_k^P(A)$ and $\Delta_k^P(A \oplus B)$. In the first step, we "$\Sigma_k^P(A)$-encode" into A all the strings $x \in K^{k+1}(A)$, where $t(n-1)+1 \leq (k+2)\cdot|x|+1 < m$. In the second step, we "$\Sigma_k^P(A)$-encode" into A all the strings $x \in K^{k+1}(A)$, where $m \leq (k+2)\cdot|x|+1 \leq t(n)$.

Let C_x be the circuit corresponding to the predicate $\tau_k(A; x0^{|x|})$, and let C_0 be the circuit corresponding to the predicate $0^m \in L_k^A$. Also, by Lemma 3.2, there is a polynomial q and a corresponding $\Delta_k(q(m))$-circuit C_n such that $C_n \lceil \rho_{A\oplus B} = 1$ if and only if $0^m \in L(M_n, K^{k-1}(A \oplus B))$. We let $t(n) = q(m)$. Note that C_n can be simplified to another $\Delta_k(q(m))$-circuit C_n' by the following replacements:

(1) all the variables v_z in the leaves of C_n having the form $z = 1z'$ are replaced by 1 if $z' \in B$ and by 0 if $z' \notin B$, and

(2) all the variables v_z of the form $z = 0z'$ are replaced by $v_{z'}$.

Thus, for any set A, and a fixed set B, $C_n \lceil_{\rho_{A\oplus B}} = C_n' \lceil_{\rho_A}$ and $C_n' \lceil_{\rho_A} = 1$ if and only if $0^m \in L(M_n, K^{k-1}(A \oplus B))$.

The requirement for stage n can be described as

$$\begin{aligned} R_n : (\exists B_0, B_1 \subseteq D(n))[&B_0 \cap B_1 = \emptyset,\ C_n' \lceil_{\rho_{B_1, B_0}} \neq *, \text{ and for all } x, \\ &t(n-1) < (k+2)|x|+1 < m,\ C_x \lceil_{\rho_{B_1, B_0}} = 1 \Leftrightarrow x \in K^{k+1}(A(n-1) \cup B_1), \\ &\text{and } C_0 \lceil_{\rho_{B_1, B_0}} = C_x \lceil_{\rho_{B_1, B_0}} = *, \text{ for all } x, m \leq (k+2)|x|+1 \leq t(n)], \end{aligned}$$

where ρ_{B_1, B_0} is the restriction that v_z is 1 if $z \in B_1$ and v_z is 0 if $z \in B_0$. Construction of A at stage n is done by the following steps.

Step 1. For all the strings x, $t(n-1) < (k+2)|x|+1 < m$, assign strings to $A(n)$ and $\overline{A(n)}$ such that $\tau_k(A; x0^{|x|})$ is true if and only if $x \in K^{k+1}(A)$.

Step 2. Replace all the variables in C_n' that are in $A(n)$ by 1 and replace those in $\overline{A(n)}$ by 0. The resulting circuit, let's still call it C_n', remains a $\Delta_k(q(m))$-circuit.

Step 3. Find a pair of finite sets B_1 and B_0 that satisfy the requirement R_n. Assign variables in B_1 to $A(n)$ and assign variables in B_0 to $\overline{A(n)}$.

Step 4. For all the strings x, $m \leq (k+2)|x|+1 \leq t(n)$, assign remaining free strings to $A(n)$ and $\overline{A(n)}$ such that $\tau_k(A; x0^{|x|})$ is true iff $x \in K^{k+1}(A)$. If there are still remaining free strings of length between $t(n-1)+1$ and $t(n)$, assign them to $\overline{A(n)}$. Go to the next stage.

The key step is Step 3. The following lemma guarantees that the requirement R_n can be satisfied at Step 3.

LEMMA 4.5. *For all $k \geq 2$, let $\{C_i\}_{i=1}^t$ be t circuits, each computing a function f_k^v, with pairwise disjoint variables. Let C be a $\Delta_k(r)$-circuit. If $t \leq 2^{\delta v^{1/6}}$ and $r \leq \delta v^{1/5}$, with*

$\delta = 1/48$, *then for sufficiently large* v, *there exists a restriction* ρ *such that* $C\lceil_\rho \neq *$ *and* $C_i\lceil_\rho = *$ *for all* $i = 1, \ldots, t$.

Proof. The proof is by induction on k. For the case of $k = 2$, the proof is similar to the proof of Theorem 3.3. Recall that in the proof of Theorem 3.3, we found a restriction ρ such that $C\lceil_\rho \neq *$ and $C_2^v\lceil_\rho = *$, where C is an $\Delta_2(r)$-circuit with $r < \delta v^{1/5}$ and C_2^v computes an f_2^v function. (See (#) in the proof of Theorem 3.3.) The number of strings that we added to the oracle set was at most $r^2 \leq (\delta v^{1/5})^2 < \sqrt{v}$. Here, every C_i, $1 \leq i \leq t$, is an OR of v AND gates, each of fanin $\sqrt{v}$. Therefore, the same restriction ρ satisfies that $C_i\lceil_\rho = *$, for all $i, 1 \leq i \leq t$.

For the induction step, consider any $k > 2$. As in the proof of Theorem 3.3, proceeding by contradiction, we need to verify that (1) with a high probability, $C\lceil_{\rho g(\rho)}$ is equivalent to a $\Delta_{k-1}(r)$-circuit, and (2) with a high probability, $C_i\lceil_{\rho g(\rho)}$ contains a subcircuit computing the function f_{k-1}^v for all $i = 1, \ldots, t$. Part (1) is identical to that in the proof of Theorem 3.3. Part (2) follows from Lemma 2.10. □

It is easy to see that C_0 contains a $C_k^{2^{m/(k+2)}}$-subcircuit computing an $f_k^{2^{m/(k+2)}}$ function. Also, for all x, $m \leq (k+2)|x| + 1 \leq t(n)$, C_x contains a subcircuit computing $f_k^{2^{m/(k+2)}}$. There are at most $2^{q(m)}$ many such C_x. To satisfy requirement R_n, we need to choose an integer m so large that $q(m) \leq (1/48)2^{m/6(k+2)}$. Thus, there exists an oracle set A such that $\Sigma_{k+1}^P(A) = \Sigma_k^P(A)$ and $\Sigma_k^P(A) \not\subseteq \Delta_k^P(A \oplus B)$, for each $k \geq 2$. This completes the proof of Theorem 4.4.

COROLLARY 4.6. *For each* $k \geq 2$, $EL_k^{P,\Delta} \subsetneq EL_k^{P,\Sigma}$.

Proof. As a point of interest, recall that $EL_2^{P,\Delta} = EL_1^{P,\Sigma}$ by Proposition 4.2. For $k = 2$, a direct modification of Baker, Gill, and Solovay's proof [BGS75] yields the existence of an oracle A such that PH(A) = NP(A) $\neq$ P($A \oplus$ SAT). For $k > 2$, just let B be SAT in Theorem 4.4. □

COROLLARY 4.7. *For each* $k \geq 1$, *there exists an oracle set A such that* $\Delta_k^P(A) \subsetneq \Sigma_k^P(A) =$ PH(A) = PSPACE(A).

Proof. Simply replace the complete set $K^{k+1}(A)$ by a complete set for PSPACE(A), and let $B = \emptyset$ in the proof of Theorem 4.4. □

Ko also constructed an oracle set relative to which the polynomial-time hierarchy has a finite number of levels and is properly contained in PSPACE [Ko89]. By combining his proof techniques with those in Theorem 4.4, we have the following corollary:

COROLLARY 4.8. *For each* $k \geq 1$, *there exists an oralce set A such that* $\Delta_k^P(A) \subsetneq \Sigma_k^P(A) =$ PH(A) $\subsetneq$ PSPACE(A).

4.2. Separating $EL_k^{P,\Delta}$ and $EL_k^{P,\Theta}$. In this section, we will separate $EL_k^{P,\Delta}$ and $EL_k^{P,\Theta}$ for each $k \geq 3$. The setup is similar to that in §4.1 for arbitrary $k \geq 2$, we will construct a set A such that PH(A) = $\Delta_k^P(A) \not\subseteq \Theta_k^P(A \oplus \text{SAT})$. As a result, $A \in EL_{k+1}^{P,\Delta}$ and $A \notin EL_{k+1}^{P,\Theta}$.

THEOREM 4.9. *For every recursive set B, and for each* $k \geq 2$, *there exists an oracle set A such that* PH(A) = $\Delta_k^P(A) \not\subseteq \Theta_k^P(A \oplus B)$.

Proof. To collapse PH to $\Delta_k^P(A)$, we will encode the $\Sigma_k^P(A)$ complete set $K^k(A)$ into a set in $\Delta_k^P(A)$. To separate $\Delta_k^P(A)$ and $\Theta_k^P(A \oplus B)$, we will use the set $L_{odd,k}^A \in \Delta_k^P(A)$ defined in the previous section:

$$L_{\text{odd},k}^A = \{x0^{kn} \mid |x| = n, \text{ and } \max\{y \mid |y| = |x|, xy0^{(k-1)n} \in L_{k-1}^A\} \text{ has odd parity}\}.$$

Now define the set

$$\hat{L}_{\text{odd},k}^A = \{x \mid \max\{y \mid |y| = |x| \text{ and } \tau_{k-1}(A; xy) \text{ is true }\} \text{ has odd parity}\},$$

where τ_{k-1} is the predicate from §4.1. Notice that for each x of length n, whether $x \in \hat{L}_{\text{odd},k}^A$ only depends on membership of strings in A of length $(k+1)n + 1$.

It is easy to verify that $\hat{L}^A_{\mathrm{odd},k}$ is in $\Delta^P_k(A)$: just use binary search querying a set in $\Sigma^P_{k-1}(A)$. Also, it is clear that, for any pair of strings x and y, whether $x \in L^A_{\mathrm{odd},k}$ and $y \in \hat{L}^A_{\mathrm{odd},k}$ are true depends on the membership of different strings in A.

The rest of the proof is very similar to that in §4.1. A is constructed by stages. The setup of $m, t(n)$, and $D(n)$ is similar to that in §3.2. At stage n of the construction, we will find an assignment of strings to A and $\overline{A}$ so that $0^m \in L^A_{\mathrm{odd},k}$ if and only if $0^m \notin L(M_n, K^{k-1}(A \oplus B))$, where M_n is the nth machine in an enumeration of $\Theta^P_k(A \oplus B)$. In stage n, we "$\Delta^P_k(A)$-encode" into A any strings $x \in K^k(A)$, $t(n-1)+1 \leq (k+1)\cdot|x|+1 \leq t(n)$, such that $x \in K^k(A) \Leftrightarrow x \in \hat{L}^A_{\mathrm{odd},k}$.

Again, let C_x be the circuit corresponding to the predicate $x \in \hat{L}^A_{\mathrm{odd},k}$, and let C_0 be the circuit corresponding to the predicate $0^m \in L^A_{\mathrm{odd},k}$. Also, by Lemma 3.6, there is a polynomial q and a corresponding $\Theta_k(q(m))$-circuit C_n such that $C_n\lceil_{\rho_{A\oplus B}}= 1$ if and only if $0^m \in L(M_n, K^{k-1}(A \oplus B))$. Let $t(n) = q(m)$. Simiar to §4.1, C_n can be simplified to a $\Theta_k(q(m))$-circuit C'_n such that $C_n\lceil_{\rho_{A\oplus B}}= C'_n\lceil_{\rho_A}$ for arbitrary set A and a fixed set B. The requirement of stage n can be described as

$$\begin{aligned}R_n : (\exists B_0, B_1 \subseteq D(n))[&B_0 \cap B_1 = \emptyset, C'_n\lceil_{\rho_{B_1,B_0}} \neq *, \text{ and for all } x,\\ &t(n-1) < (k+1)|x|+1 < m, C_x\lceil_{\rho_{B_1,B_0}} = 1 \Leftrightarrow x \in K^k(A(n-1) \cup B_1),\\ &\text{and } C_0\lceil_{\rho_{B_1,B_0}} = C_x\lceil_{\rho_{B_1,B_0}} = *, \text{ for all } x, m \leq (k+1)|x|+1 \leq t(n)],\end{aligned}$$

where ρ_{B_1,B_0} is the restriction that v_z is 1 if $z \in B_1$ and v_z is 0 if $z \in B_0$. Steps in construction of A at stage n are essentially the same as that in §4.1. The following lemma ensures that the requirement R_n can be satisfied.

LEMMA 4.10. *For all $k \geq 2$, let $\{C_i\}^t_{i=1}$ be t circuits, each computing an h^v_k function, with pairwise disjoint variables. Let C be a $\Theta_k(r)$-circuit. If $t \leq 2^{\delta v^{1/6}}$ and $r \leq \delta v^{1/5}$, with $\delta = 1/24$, then for sufficiently large n, there exists a restriction ρ such that $C\lceil_\rho \neq *$ and $C_i\lceil_\rho = *$ for all $i = 1, \ldots, t$.*

Proof. Again, the proof is by induction on k. The base case of $k = 2$ is proved similarly to the proof of Theorem 3.7. Recall that in **Process FF**, we flip-flopped the output of an H^v_k circuit computing an h^v_k function by adding to the oracle set the variables in the leftmost undecided OR gate of H^v_k. We then find a restriction ρ such that $C\lceil_\rho \neq *$ and $H^v_2\lceil_\rho = *$ (see (∗) in the proof of Theorem 3.7). Here we have t many H^v_k circuits. We can easily modify **Process FF** such that at each step of flip-flop we add to the oracle set the variables of the leftmost undecided OR gate of C_i, for all $1 \leq i \leq t$. It is not difficult to see that, by a similar argument, we can find a restriction ρ such that $C\lceil_\rho \neq *$ and $C_i\lceil_\rho = *$ for all $1 \leq i \leq t$.

For the induction step, consider any $k > 2$. As in the proof of Theorem 3.7, proceeding by contradiction, we need to verify that (1) with a high probability, $C\lceil_{\rho g(\rho)}$ is equivalent to a $\Theta_{k-1}(r)$-circuit, and that (2) with a high probability, $C_i\lceil_{\rho g(\rho)}$ contains a subcircuit computing the function h^v_{k-1} for all $i = 1, \ldots, t$. Part (1) is identical to that in the proof of Theorem 3.7 and part (2) is proved by Lemma 2.10. □

Note that C_0 contains an $H^{2^{m/(k+1)}}_k$- subcircuit computing a function $h^{2^{m/(k+1)}}_k$. Also, for all $x, m \leq (k+1)|x|+1 \leq t(n)$, C_x contains a subcircuit computing $h^{2^{m/(k+1)}}_k$. There are at most $2^{q(m)}$ many such C_x. To satisfy requirement R_n, we need to choose an integer m so large that $q(m) \leq (1/24)2^{m/6(k+1)}$. Thus, there exists an oracle set A such that $\Sigma^P_k(A) = \Delta^P_k(A)$ and $\Delta^P_k(A) \not\subseteq \Theta^P_k(A \oplus B)$, for each $k \geq 2$. This completes the proof of Theorem 4.9.

COROLLARY 4.11. *For each $k \geq 3$, $EL^{P,\Theta}_k \subsetneq EL^{P,\Delta}_k$.*

Proof. Simply replace B by SAT in the proof of Theorem 4.9. □

COROLLARY 4.12. *For each $k \geq 2$, there exists an oracle set A such that $\Theta^P_k(A) \subsetneq \Delta^P_k(A) =$* PH$(A) =$ PSPACE(A).

Proof. Simply replace $K^k(A)$ by a complete set for PSPACE(A) and replace B by the empty set in the proof of Theorem 4.9. □

COROLLARY 4.13. *For each* $k \geq 2$, *there exists an oracle set A such that* $\Theta_k^P(A) \subsetneq \Delta_k^P(A) = \mathrm{PH}(A) \subsetneq \mathrm{PSPACE}(A)$.

4.3. Separating $EL_k^{P,\Theta}$ and $EL_{k-1}^{P,\Sigma}$. In this section, we will find a recursive set A that is in $EL_k^{P,\Theta}$ but not in $EL_{k-1}^{P,\Sigma}$ for each $k \geq 3$. The setup is similar to that in §§4.1 and 4.2: for each $k \geq 2$, we construct a set A such that $\mathrm{PH}(A) = \Theta_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus \mathrm{SAT})$. Thus, A is in $EL_{k+1}^{P,\Theta}$ and not in $EL_k^{P,\Sigma}$. We first prove the following theorem.

THEOREM 4.14. *For every recursive set B, for each* $k \geq 2$, *there exists a recursive set A such that* $\mathrm{PH}(A) = \Theta_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus B)$.

Proof. Recall that $L_{\mathrm{xor},k}^A = \{x0^{kn} \mid |x| = n$, and exactly one of $0^n x 0^{(k-1)n}$ or $1^n x 0^{(k-1)n}$ is in $L_{k-1}^A\}$. Define the following language:

$$\hat{L}_{\mathrm{xor},k}^A = \{x \mid \text{ exactly one of } \tau_{k-1}(A; 0^{|x|}x) \text{ or } \tau_{k-1}(A; 1^{|x|}x) \text{ is true } \},$$

where τ_k is defined in the previous section. It is easy to see that both $L_{\mathrm{xor},k}^A$ and $\hat{L}_{\mathrm{xor},k}^A$ are in $\Theta_k^P(A)$. A will be constructed so that $L_{\mathrm{xor},k}^A \notin \Sigma_{k-1}^P(A \oplus B)$ and so that for all x, $x \in K^k(A) \Leftrightarrow x \in \hat{L}_{\mathrm{xor},k}^A$. It will follow that $\mathrm{PH}(A) = \Theta_k^P(A)$ and $\Theta_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus B)$.

The setup of the construction of A, as well as m, $t(n)$, and $D(n)$ are the same as in §§4.1 and 4.2. At stage n of the construction, we will ensure that for all x, $t(n-1) < (k+1)|x|+1 \leq t(n)$, $x \in K^k(A) \Leftrightarrow x \in \hat{L}_{\mathrm{xor},k}^A$. Let C_0 be the circuit corresponding to the predicate $0^m \in L_{\mathrm{xor},k}^A$ and for all x, $t(n-1) < (k+1)|x|+1 \leq t(n)$, let C_x represent the circuit corresponding to the predicate $x \in \hat{L}_{\mathrm{xor},k}^A$. Let $\sigma_1, \sigma_2, \ldots$ be an enumeration of $\Sigma_{k-1}^{P,1}$-predicates. By Lemma 2.3, there exist a polynomial q and a $\Sigma_{k-1}(q(m))$-circuit C_n corresponding to the nth predicate σ_n. We let $t(n) = q(m)$. Using the same replacement technique as in the §4.1, C_n can be simplified to a Σ_{k-1}-circuit C_n' such that $C_n \lceil_{\rho_{A\oplus B}} = C_n' \lceil_{\rho_A}$ for arbitrary set A and a fixed set B. The requirement R_n of stage n of the construction is

$$\begin{aligned} R_n : (\exists B_0, B_1 \subseteq D(n))[&B_0 \cap B_1 = \emptyset, C_n' \lceil_{\rho_{B_1,B_0}} \neq *, \text{for all } x, \\ &t(n-1) < (k+1)|x| + 1 < m, C_x \lceil_{\rho_{B_1,B_0}} = 1 \Leftrightarrow x \in K^k(A(n-1) \cup B_0), \\ &\text{and } C_0 \lceil_{\rho_{B_1,B_0}} = C_x \lceil_{\rho_{B_1,B_0}} = *, \text{ for all } x, m \leq (k+1)|x| + 1 \leq t(n)], \end{aligned}$$

where ρ_{B_1,B_0} is the restriction that v_z is 1 if $z \in B_1$ and v_z is 0 if $z \in B_0$. The steps in the construction of A at stage n are essentially the same as that in §4.1. We need the following lemma to ensure that R_n will can be satisfied at stage n.

LEMMA 4.15. *For all* $k \geq 2$, *let* $\{C_i\}_{i=1}^t$ *be t circuits, each computing an* l_k^v *function, with pairwise disjoint variables. Let C be a* $\Sigma_{k-1}(r)$*-circuit. If* $t \leq 2^{\delta v^{1/6}}$ *and* $r \leq \delta v^{1/3}$, *with* $\delta = 1/12$, *then for sufficiently large n, there exists a restriction* ρ *such that* $C\lceil_\rho \neq *$ *and* $C_i \lceil_\rho = *$ *for all* $i = 1, \ldots, t$.

Proof. The proof is by induction on k. To prove the base case of $k = 2$, let us consider circuits C and C_i for all $i = 1, \ldots, t$. C is an OR of AND's with bottom fanin $\leq r$. C_i is an XOR of two subcircuits, each computing an f_1^v function. Note that all the leaves of all C_i's are unique positive variables. Let v_i be the leftmost variable in C_i, $1 \leq i \leq t$, and let $V = \{v_i \mid 1 \leq i \leq t\}$.

Let F denote the variables that are assigned 1, and let $\overline{F}$ denote the variables that are assigned 0. Initially, both F and $\overline{F}$ are empty. We will first present the algorithm for constructing F and $\overline{F}$ and then show that the restriction $\rho_{F,\overline{F}}$ is the desired restriction. Recall that $\rho_\emptyset$ is the restriction that assigns 0 to all variables.

```
Process F:
        $F = \overline{F} = \emptyset$.
        if $C\lceil_{\rho_{\emptyset}} = 1$ then
                Choose one AND gate of $C$ with all negative variables; Assign
                all the variables to $\overline{F}$
        else
                if there is an $S \subseteq V$ such that $C\lceil_{\rho_S} = 1$ then
                        Select such an $S$;
                        Choose one AND gate $D$ of $C$ such that $D\lceil_{\rho_S} = 1$;
                        Assign all negative variables in $D$ to $\overline{F}$ and assign all positive
                        variables in $D$ to $F$
                else
                        $F = \emptyset, \overline{F} = \Sigma^* - V$.
                end(* if *)
        end(* if *)
```

We now argue the correctness of **Process F**. Consider the first case when $F = \emptyset$ and $C\lceil_{\rho_{\emptyset}} = 1$. Since C is a circuit of an OR of AND gates, there must exist an AND gate D in C such that D contains only negative variables. Put all the variables in D into $\overline{F}$ to preserve the the output of C. We have $C\lceil_{\rho_{F,\overline{F}}} = 1$. Note that the number of variables in D is $\leq r \leq \delta v^{1/3} < \sqrt{v}$, thus $C_i\lceil_{\rho_{F,\overline{F}}} = *$, for all $i = 1, \ldots, t$. In this case, $\rho_{F,\overline{F}}$ is the desired restriction.

On the other hand, if $C\lceil_{\rho_{\emptyset}} = 0$, then we choose some j many different v_i's and put these variables into S. Suppose, for all the different choices of S, $C\lceil_{\rho_S}$ remains 0. Then we know that for all 2^t possible S, $C\lceil_{\rho_S} = 0$, and, for each C_i, $C_i\lceil_{\rho_S} = 1$ if and only if $v_i \in S$. This implies that we can arbitrarily force any combination of C_i's to output 1 or 0, by choice of S, without changing the output of C. Thus, $C\lceil_{\rho_{\emptyset,\Sigma^*-V}} = 0$ and $C_i\lceil_{\rho_{\emptyset,\Sigma^*-V}} = *$, for all $1 \leq i \leq t$. Let $F = \emptyset$ and $\overline{F} = \Sigma^* - V$ and $\rho_{F,\overline{F}}$ is the desired restriction.

The last case is that there is an S such that $C\lceil_{\rho_S} = 1$. In this case, fix an AND gate D of C such that $D\lceil_{\rho_S} = 1$. We reserve the output of C by assigning the negative variables in D to $\overline{F}$ and the positive variables in D to F. Note that each positive variable in D is in S since D is an AND gate and $D\lceil_{\rho_S} = 1$. Also important is that for any restriction ρ_S, there is only one positive variable in C_i that is assigned 1 by ρ_S. The number of variables in $\overline{F}$ is bounded by $r \leq \delta v^{1/3} < \sqrt{v}$. Thus, $C\lceil_{\rho_{F,\overline{F}}} = 1$ and $C_i\lceil_{\rho_{F,\overline{F}}} = *$ for all $1 \leq i \leq t$. In summary, there is a pair of finite sets F and $\overline{F}$ such that $F \cap \overline{F} = \emptyset$, $C\lceil_{\rho_{F,\overline{F}}} \neq *$, and $C_i\lceil_{\rho_{F,\overline{F}}} = *$ for all i, $1 \leq i \leq t$.

Finally, the induction step is proved by Lemma 2.7 and Lemma 2.10. □

Note that C_0 contains a subcircuit computing an $l_k^{2^{m/(k+1)}}$ function. Also, for all x, $m \leq (k+1)|x| + 1 \leq t(n)$, C_x contains a subcircuit computing an $l_k^{2^{m/(k+1)}}$ function. To satisfy requirement R_n, we only need to choose an integer m so large that $t(n) \leq (1/12)2^{m/6(k+1)}$. We then conclude that there exists an oracle set A such that $\Sigma_k^P(A) = \Theta_k^P(A)$ and $\Theta_k^P(A) \not\subseteq \Sigma_{k-1}^P(A \oplus B)$, for each $k \geq 2$. This completes the proof of Theorem 4.14.

COROLLARY 4.16. *For each* $k \geq 3$, $EL_{k-1}^{P,\Sigma} \subsetneq EL_k^{P,\Theta}$.

Proof. Simply replace B by SAT in the proof of Theorem 4.14. □

COROLLARY 4.17. *For each* $k \geq 2$, *there exists an oracle set* A *such that* $\Sigma_{k-1}^P(A) \subsetneq \Theta_k^P(A) = \mathrm{PH}(A) = \mathrm{PSPACE}(A)$.

Proof. Simply replace $K^k(A)$ by a complete set for PSPACE(A) and replace B by the empty set in the proof of Theorem 4.14. □

COROLLARY 4.18. *For each* $k \geq 2$, *there exists an oracle set* A *such that* $\Sigma_{k-1}^P(A) \subsetneq \Theta_k^P(A) = \mathrm{PH}(A) \subsetneq \mathrm{PSPACE}(A)$.

Acknowledgment. The authors are grateful to the anonymous referees for their comments.

REFERENCES

[AH92] E. Allender and L. Hemachandra, *Lower bounds for the low hierarchy*, J. Appl. Comput. Mach., 39 (1992), pp. 234–251.

[BBS86] J. Balcázar, R. Book, and U. Schöning, *Sparse sets, lowness, and highness*, SIAM J. Comput., 15 (1986), pp. 739–747.

[BGS75] T. Baker, J. Gill, and R. Solovay, *Relativizations of the P =? NP-question*, SIAM J. Comput., 4 (1975), pp. 431–441.

[FSS81] M. Furst, J. Saxe, and M. Sipser, *Parity, circuits, and the polynomial-time hierarchy*, in Proc. 22th Annual IEEE Symposium on Foundations of Computer Science, 1981, pp. 260–270.

[Has87] J. D. Hastad, *Computational Limitations for Small-depth Circuits*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[Ko89] K. Ko, *Relativized polynomial time hierarchies having exactly k levels*, SIAM J. Comput., 18 (1989), pp. 392–408.

[Ko91] ———, *Separating the low and high hierarchies by oracles*, Inform. and Comput., 90 (1991), pp. 156–177.

[LS91] T. Long and M. Sheu, *A refinement of the low and high hierarchies*, Tech. report OSU-CISRC-2/91-TR6, The Ohio State University, Columbus, OH, 1991.

[Sch83] U. Schöning, *A low and a high hierarchy within NP*, J. Comput. System Sci., 27 (1983), pp. 14–28.

[Sip83] M. Sipser, *Borel sets and circuit complexity*, in Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 61–69.

[Sto76] L. Stockmeyer, *The polynomial-time hierarchy*, Theor. Comput. Sci., 3 (1976), pp. 1–22.

[Wra76] C. Wrathall, *Complete sets and the polynomial hierarchy*, Theor. Comput. Sci., 3 (1976), pp. 23–33.

[Yao85] A. Yao, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 1–10.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 510–519, June 1994

004

COMPLEXITY OF NETWORK RELIABILITY AND OPTIMAL RESOURCE PLACEMENT PROBLEMS*

DONALD B. JOHNSON† AND LARRY RAAB‡

Abstract. A fundamental problem of distributed system design in an existing network where components can fail is finding an optimal location at which to place a resource. This paper proves exactly how hard this placement problem is under the measure of data availability. Specifically, it shows that the optimal placement problem for availability is #*P*-complete, a measure of intractability at least as severe as *NP*-completeness. To obtain these results, the environment in which a distributed system operates is modelled by a *probabilistic graph*, which is a set of fully reliable vertices representing sites and a set of edges representing communication links, each operational with a rational probability. Finding the optimal placement in a probabilistic graph is proved to be #*P*-complete by giving a sequence of Turing reductions from #Satisfiability. This result is generalized to networks in which each site and each link has an independent, rational operational probability and to networks in which all the sites or all the links have fixed, uniform operational probabilities. Given the anticipated computational difficulty of finding an exact solution, the requirements for effective, practical approximation methods are discussed.

Key words. #*P* complexity class, resource placement, database, simulation, networks, reliability graph

AMS subject classifications. 68Q25, 68P15, 68M15

1. Introduction.

Determining the optimal placement of a resource, be it a file, database, or data object, is one of the most well-studied problems in computer science. Research into the "file assignment problem," or FAP as it is now known, [3], [8], dates back to Chu in 1969 [4] and even earlier when viewed as the single commodity warehouse problem (see, for example, [19]).

Our interest in this placement problem is motivated by our work with database replica control protocols [15], [17], [16]. These protocols attempt to increase the accessibility of a data object by replicating that object throughout the network. Our work has shown that, given the database consistency constraints, there is a nontrivial bound on the benefits of replication over an optimally located nonreplicated data object [17]. Thus, it is natural to attempt a complexity characterization, which we present in this paper.

The most general form of the optimal placement problem is as follows: given a set of sites, communication links, rational reliability probabilities on both the sites and links, and a distribution of access requests, find the *optimal* site. A site x is **optimal** if and only if placing the resource at site x maximizes *availability*. **Availability** is defined as the probability that an access request, when submitted according to an access request distribution, arrives at a site that can communicate with site x. A rational reliability ρ for a site (and similarly for a link) is the steady-state probability that the site is operational. Therefore, $\rho = \frac{\text{MTTF}}{\text{MTTF+MTTR}}$, where MTTF is the mean time to failure for a site and MTTR is the mean time to recovery for a site.

Thus, the optimal location at which to place the sole copy of a resource in a distributed environment is a function of the network topology, the site and link reliabilities, and the access request distribution. We show that since the underlying graph reliability problems are #*P*-complete, so also is this **optimal placement** problem. #*P*-complete implies, among other things, that an efficient (polynomial) solution to this problem can be found only if $P = NP$ [22]. In this paper we prove that the simplified problem, where the sites are infallible, links operate with probability ρ, and the access request distribution is uniform, is #*P*-complete.

*Received by the editors May 26, 1992; accepted for publication (in revised form) November 30, 1992.

†Department of Mathematics and Computer Science, Dartmouth College, Bradley Hall, Hanover, New Hampshire 03755 (djohnson@cs.dartmouth.edu).

‡Transarc Corporation, The Gulf Tower, 707 Grant Street, Pittsburgh, Pennsylvania 15219 (raab@transarc.com).

We call this the **simplified model** and call the graph representing such a network a **probability graph**. Using the simple technique of restriction [10], this result generalizes to include networks in which each site and each link has an independent, rational operational probability, to networks with fixed, uniform site reliabilities, and to arbitrary access request distributions.

We begin by listing each of the problems that we use to prove our #*P*-completeness result. Each of these problems is an interesting probability graph problem in its own right. In §4, we prove that each of these problems are #*P*-complete. We also generalize the main complexity result to include classes of networks with nonuniform link reliabilities and networks with uniform, fixed site reliabilities. This latter class includes such networks as radio broadcast networks [1] and single-bus networks like Ethernet. Having proven this problem computationally difficult, we have little hope of finding an efficient, exact solution. Therefore, we conclude with a discussion of practical methods for approximating the optimal copy placement.

2. Related work.

As mentioned above, determining the optimal placement is one of the most well-studied problems in computer science. Our work differs from all others of which we are aware in that we wish to optimize the availability measure, the probability that an access request arrives at a site that can contact the resource. In addition, we show that this problem is $\#P$-complete, not NP-complete as is frequently shown for other location problems, and is, therefore, at least as "hard" as NP-complete problems and may well be "harder" [10].

In [7], Dowdy and Foster present a survey of research dealing with FAP, including a description of fourteen models and a list of twenty-one others. More recent models with approximate solutions are discussed in [11]. Although the models vary considerably, they all attempt to minimize some cost measure (such as storage or communication cost) or maximize throughput. Although some of these models include an availability constraint, they neither maximize availability nor define it as we have here.

Related work also appears in the context of network reliability computations. In 1965 Kiryukhin investigated the optimal placement of a given number of communication links relative to the availability measure. Having simplified the problem by considering only paths of length less than three, Kiryukhin presented a nonlinear programming solution. See [9] and [25] for more detailed discussions.

Rosenthal provides the first NP-hardness result for these network reliability problems [20]. His proof reduces the Steiner Tree problem to the computation of the probability that a specified subset of a graph is connected. NP-hardness has also been shown for determining whether two sites are connected [24] and whether all sites are connected [13], [18]. A survey of these results and related approximation algorithms can be found in [5].

3. Problem definitions.

In this section, we define each of a sequence of combinatorial problems that we use to prove that finding the optimal location of a single copy is #*P*-complete. In [23] Valiant defined the class of problems called #*P* and showed #SAT to be #*P*-complete using a modification of Cook's construction for SAT [6]. In [24] Valiant showed that CONNECTEDNESS is also #*P*-complete. The other three problems are shown to be #*P*-complete with respect to Turing reductions in §4. In Table 1 we define the terms and notation we use.

We maximize availability by maximizing $\mathcal{E}\,[v]$, the expected size of the group containing a site v. $\mathcal{E}\,[v]/n$ is the availability achieved on a network with n sites and a single copy located at site v, since, in this simplified model, access requests are submitted uniformly at random, and only requests submitted to sites within the group containing v will be granted. Therefore, site v is an **optimal location** if and only if $\mathcal{E}\,[v] \geq \mathcal{E}\,[u]$ for all sites u.

In the problems that follow, we use the phrase "the expected group size of vertex v," meaning the expected number of sites in the group containing v. Also, if more than one vertex

TABLE 1

Notation and Terminology

- $G = (V, E)$ is a graph of vertices V and undirected edges E. G represents a network with sites V and bidirectional communication links E.
- n and m denote $|V|$ and $|E|$, respectively, when $G = (V, E)$ is clear from the context.
- $\rho = p/q$ is the rational link reliability, where $0 \le \rho \le 1$ and $\gcd(p, q) = 1$.
- A **group** is a maximal set of connected vertices. Groups comprising a probability graph change over time due to component failures and recoveries.
- $\mathcal{E}_G[v]$ is the expected size of the group (i.e., number of vertices in the group) containing vertex v in probability graph G. We will simply say $\mathcal{E}[v]$ when G is clear from the context.
- $\mathcal{P}r(S)$ is the probability that the logical statement S is true.
- $\mathcal{P}r(S|T)$ is the conditional probability that the logical statement S is true given that the logical statement T is true.
- $\overline{S}$ is the negation of the logical statement S; therefore, $\mathcal{P}r(\overline{S}) = 1 - \mathcal{P}r(S)$.
- $c(u, w)$ is the logic statement "vertices u and w are connected." The probability graph containing u and w will be clear from context.

has maximal expected group size, OPTLOC may return any one of these vertices.

1. **#SAT** (#SAT)
   INSTANCE: A logical formula F in n variables.
   QUESTION: How may different truth assignments that satisfy F are there to the n variables?
2. **CONNECTEDNESS** (CON)
   INSTANCE: A probability graph $G = (V, E)$ and vertices $v_1, v_2 \in V$.
   QUESTION: What is the probability that vertices v_1 and v_2 are connected?
3. **EXPECTED SIZE** (EXPSZ)
   INSTANCE: A probability graph $G = (V, E)$ and vertex $v \in V$.
   QUESTION: What is the expected group size of vertex v?
4. **BOUNDED EXPECTED SIZE** (BEXPSZ)
   INSTANCE: A probability graph $G = (V, E)$, vertex $v \in V$, and a rational number B.
   QUESTION: Has v expected group size greater than or equal to B?
5. **OPTIMAL LOCATION** (OPTLOC)
   INSTANCE: A probability graph $G = (V, E)$.
   QUESTION: Which $v \in V$ has the largest expected group size?

4. Reductions. In this section, we either prove or cite proofs for each of the problems defined in the previous section. The first two problems are covered elsewhere and citations are given. The remaining three problems are shown to be *#P*-complete. We include a subsection with two related lemmas that are used in §4.3. Again, see Table 1 for an explanation of the terms and notation.

4.1. Preliminary reductions.

THEOREM 4.1. #SAT *is #P-complete.*

Proof. In [6] Cook proved that SAT is *NP*-complete. Valiant modified Cook's proof to show that #SAT is *#P*-complete [23]. □

THEOREM 4.2. CON *is #P-complete.*

Proof. A reduction from #SAT to CON is given by Valiant in [24]. □

THEOREM 4.3. EXPSZ *is #P-complete.*

Proof. Let $G = (V, E)$ be a probability graph. Then it is not difficult to show that the expected size of the group containing $v \in V$, $\mathcal{E}[v]$, is equal to $\sum_{w \in V} \mathcal{P}r\big(c(v, w)\big)$. Thus

EXPSZ is in $\#P$ since we can solve EXPSZ with $|V|$ queries to an CON oracle, and CON is in $\#P$.

We show that EXPSZ is $\#P$-hard using a Turing reduction from EXPSZ to CON. We solve CON by calculating the expected group size of a vertex in each of two networks.

Let $G = (V, E)$ and $u, v \in V$ be an instance of CON.

Let $G' = (V', E')$, where $V' = V \cup \{u'\}$ and $V' = V \cup \{(u, u')\}$, where u' is a new vertex not in V. Then

$$\begin{aligned}\mathcal{E}_{G'}[v] &= \sum_{w\in V'} \mathcal{P}r\big(c(v,w)\big)\\ &= \mathcal{P}r\big(c(v,u')\big) + \sum_{w\in V} \mathcal{P}r\big(c(v,w)\big)\\ &= \rho\, \mathcal{P}r\big(c(v,u)\big) + \sum_{w\in V} \mathcal{P}r\big(c(v,w)\big)\\ &= \rho\, \mathcal{P}r\big(c(v,u)\big) + \mathcal{E}_G[v].\end{aligned}$$

Therefore, $\mathcal{P}r\big(c(v,u)\big) = \frac{1}{\rho}\Big(\mathcal{E}_{G'}[v] - \mathcal{E}_G[v]\Big)$, and calculating $\mathcal{E}_G[v]$ must be $\#P$-complete since calculating $\mathcal{P}r\big(c(v,u)\big)$ is $\#P$-complete. □

THEOREM 4.4. BEXPSZ *is #P-complete.*

Proof. Clearly BEXPSZ is in $\#P$ since we can solve BEXPSZ with one query to an EXPSZ oracle, and EXPSZ is in $\#P$.

We show that BEXPSZ is $\#P$-hard using a Turing reduction from EXPSZ to BEXPSZ.

Let $G = (V, E)$ and vertex $v \in V$ be an instance of the EXPSZ problem, that is, we wish to determine C, the expected size of the group containing vertex v. Let $n = |V|$ and $m = |E|$. Then there are 2^m possible graph states, and the probability of any one state with k operational links, $0 \leq k \leq m$, is $\rho^k(1-\rho)^{m-k}$. Therefore, C, $1 \leq C \leq n$, is a multiple of $1/q^m$ and is one of nq^m possible values. (Recall from Table 1 that q is an integer equal to $p \cdot \rho$.) Suppose, then, that we have an oracle that can solve BEXPSZ. Then we can use a binary search procedure to query this oracle until we find the exact value of C. This can be done in $a \leq \lceil \log(nq^m) \rceil$ queries. Since $m \leq n(n-1)/2$, $a \leq \lceil \log(nq^{n(n-1)/2}) \rceil = O(n^2)$.

Since EXPSZ is $\#P$-complete, and since we can solve EXPSZ with a polynomial number of queries to a BEXPSZ oracle, it must be that BEXPSZ is $\#P$-hard. □

4.2. Related lemmas. We simplify the task of proving that OPTLOC is $\#P$-complete by establishing two lemmas. The first lemma states that the expected group size of vertex v in graph G, $\mathcal{E}_G[v]$, is at least as large as the expected group size of any other vertex u, $\mathcal{E}_G[u]$, times the probability the v and u are connected.

LEMMA 4.5. *Let $G = (V, E)$ be a probability graph and $u, v \in V$. Then $\mathcal{E}_G[v] \geq \mathcal{P}r\,(c(v,u))\ \mathcal{E}_G[u]$.*

Proof.

$$\begin{aligned}\mathcal{E}_G[v] &= \textstyle\sum_{w\in V} \mathcal{P}r(c(v,w))\\ &\geq \textstyle\sum_{w\in V} \mathcal{P}r(c(v,u) \text{ and } c(u,w))\\ &= \mathcal{P}r(c(v,u))\ \textstyle\sum_{w\in V} \mathcal{P}r(c(u,w) \mid c(v,u))\\ &\geq \mathcal{P}r(c(v,u))\ \textstyle\sum_{w\in V} \mathcal{P}r(c(u,w))\\ &= \mathcal{P}r(c(v,u))\ \mathcal{E}_G[u]. \quad \square\end{aligned}$$

The following lemma states that, for any integer $c \geq |V|$, we can make any vertex $v \in V$ the *optimal* vertex by adding $\lceil (c+1)/\rho \rceil$ vertices, each adjacent to v.

LEMMA 4.6. *Let $G = (V, E)$ be a probability graph and $v \in V$. Let $G' = (V', E')$, where $V' = V \cup \{x_i \mid 1 \leq i \leq \lceil (c+1)/\rho \rceil\}$ and $E' = E \cup \{(v, x_i) \mid 1 \leq i \leq 2c+2\}$. For any integer $c \geq |V|$, v is the unique optimal vertex in G'.*

Proof. Let w be some vertex in $V - v$.

$$\begin{aligned} \mathcal{E}_{G'}[v] &= \mathcal{E}_G[v] + \rho\lceil (c+1)/\rho \rceil \quad \text{since link reliabilities} = \rho. \\ &\geq \mathcal{P}r(c(v,w))\,\mathcal{E}_G[w] + \rho\lceil (c+1)/\rho \rceil \quad \text{by 4.5.} \\ &> \mathcal{E}_G[w] + \mathcal{P}r(c(v,w))\,\rho\lceil (c+1)/\rho \rceil \\ &\qquad \text{since } 0 \leq \mathcal{P}r(c(v,w)) \leq 1 \text{ and } \mathcal{E}_G[w] < \rho\lceil (c+1)/\rho \rceil. \\ &= \mathcal{E}_{G'}[w] \quad \text{since all paths from } w \text{ to any } x_i \text{ pass though } v. \qquad \square \end{aligned}$$

4.3. Primary reduction. In this section, we use the previous reductions and lemmas to prove that optimally placing a single copy is $\#P$-complete.

THEOREM 4.7. OPTLOC *is $\#P$-complete.*

Proof. Clearly OPTLOC is in $\#P$ since we can solve OPTLOC using one query to an EXPSZ oracle for each $v \in V$, and EXPSZ is in $\#P$.

We show that OPTLOC is $\#P$-hard using a polynomial time reduction from BEXPSZ to OPTLOC, that is we show that we can solve the BEXPSZ problem using a machine for solving the OPTLOC problem.

Let $G_v = (V_v, E_v)$, A, $v \in V_v$ be an instance of the BEXPSZ problem, with $n = |V_v|$ and $m = |E_v|$. We will use OPTLOC to determine in polynomial time whether or not $\mathcal{E}[v] \geq A$.

We know that $\mathcal{E}[v] = \sum_{k=0}^{m} d_k \rho^k (1-\rho)^{m-k}$, where each d_k is the sum of the sizes of the groups containing site v in all states with exactly k operational links. This can be rewritten as $\sum_{k=0}^{m} \sum_{j=0}^{m-k} \binom{m-k}{j} d_k (-1)^{m-k-j} \rho^{m-j}$, using the binomial theorem. If we subtract 1 (since site v is always operational), and we subtract $D'\rho$ for as large an integer D' as possible, we are left with a positive rational number D'' less than ρ. Thus we can rewrite $\mathcal{E}[v]$ as $1 + D'\rho + \sum_{i=2}^{m} d_i \rho^i$, where each d_i is a nonnegative integer less than q (i.e., $\sum_{i=2}^{m} d_i \rho^i$ is the base-ρ expansion of D'').

We would like to express A in the same manner, as 1 plus $A'\rho$ plus a base ρ expansion of $A - 1 - A'\rho$. But this expansion may not terminate in base-ρ. Instead we define B, a terminating approximation of A, such that $\mathcal{E}[v] \geq A$ if and only if $\mathcal{E}[v] \geq B$. We form B simply by truncating A after the mth place and adding ρ^m if $B \neq A$. Thus for some sequence of positive integers b_i each less than q,

$$B = 1 + B'\rho + \sum_{i=2}^{m} b_i \rho^i.$$

We give the reduction below, an explanation following the reduction, and an example in Fig. 1.

4.3.1. Reduction.

Let $G_u = (V_u, E_u)$ where

$V_u = \{u\} \cup \{u_i | 1 \leq i \leq B'\}$,

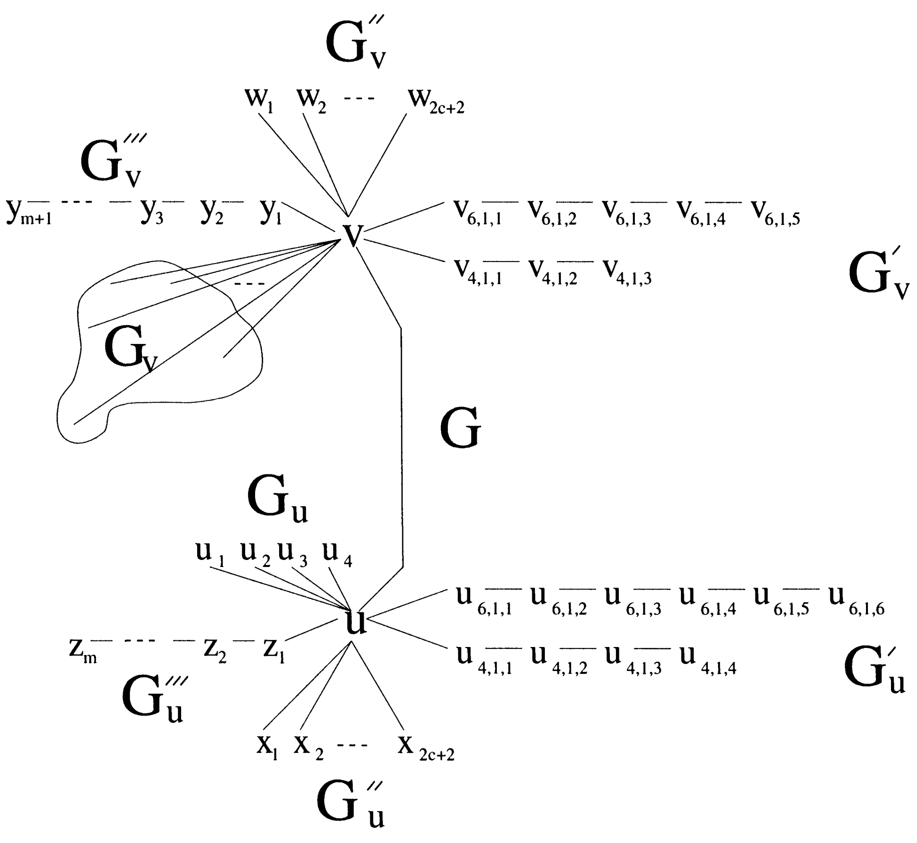
$E_u = \{(u, u_i) | 1 \leq i \leq B'\}$.

Let $G'_v = (V'_v, E'_v)$ where

$V'_v = V_v \cup \{v_{i,j,k} | 1 < i \leq m \text{ and } 1 \leq j \leq b_i \text{ and } 1 \leq k \leq i-1\}$,

$E'_v = E_v \cup \{(v, v_{i,j,1}) | 1 < i \leq m \text{ and } 1 \leq j \leq b_i\}$

$\cup \{(v_{i,j,k}, v_{i,j,k+1}) | 1 < i \leq m \text{ and } 1 \leq j \leq b_i \text{ and } 1 \leq k < i-1\}$.


G''_v
w_1 w_2 --- w_{2c+2}
G'''_v
y_{m+1} --- y_3 y_2 y_1
v
$v_{6,1,1}$ $v_{6,1,2}$ $v_{6,1,3}$ $v_{6,1,4}$ $v_{6,1,5}$
$v_{4,1,1}$ $v_{4,1,2}$ $v_{4,1,3}$
G'_v

G_v
G
G_u
u_1 u_2 u_3 u_4
u
$u_{6,1,1}$ $u_{6,1,2}$ $u_{6,1,3}$ $u_{6,1,4}$ $u_{6,1,5}$ $u_{6,1,6}$
z_m --- z_2 z_1
$u_{4,1,1}$ $u_{4,1,2}$ $u_{4,1,3}$ $u_{4,1,4}$
G'_u
G'''_u
x_1 x_2 --- x_{2c+2}
G''_u

FIG. 1. *This figure represents the graph G given an initial graph G_v with $\rho = \frac{1}{2}$ and $B = 3\frac{5}{64}$. (Therefore, $B' = 4$, $b_1 = b_2 = b_3 = b_5 = 0$, and $b_4 = b_6 = 1$.) The name of each of the intermediate graphs is given near the portion of G that was introduced by that intermediate graph. (Note that vertex v is in G_v, although this is not evident from the figure.)*

Let $G'_u = (V'_u, E'_u)$ where
$V'_u = V_u \cup \{u_{i,j,k} \mid 1 < i \leq m$ and $1 \leq j \leq b_i$ and $1 \leq k \leq i\}$,
$E'_u = E_u \cup \{(u, u_{i,j,1}) \mid 1 < i \leq m$ and $1 \leq j \leq b_i\}$
$\cup\{(u_{i,j,k}, u_{i,j,k+1}) \mid 1 < i \leq m$ and $1 \leq j \leq b_i$ and $1 \leq k < i\}$.

Let $c = \max(\ |V'_v|\ ,\ |V'_u|\ )$.

Let $G''_v = (V''_v, E''_v)$ where
$V''_v = V'_v \cup \{w_i \mid 1 \leq i \leq \lceil (c+1)/\rho \rceil\}$,
$E''_v = E'_v\{(v, w_i) \mid 1 \leq i \leq \lceil (c+1)/\rho \rceil\}$.

Let $G''_u = (V''_u, E''_u)$ where
$V''_u = V'_u \cup \{x_i \mid 1 \leq i \leq \lceil (c+1)/\rho \rceil\}$,
$E''_u = E'_u \cup \{(u, x_i) \mid 1 \leq i \leq \lceil (c+1)\rho \rceil\}$.

Let $G'''_v = (V'''_v, E'''_v)$ where
$V'''_v = V''_v \cup \{y_i \mid 1 \leq i \leq m+1\}$,
$E'''_v = E''_v \cup \{(y_i, y_{i+1}) \mid 1 \leq i \leq m\} \cup \{(v, y_1)\}$.

Let $G_u''' = (V_u''', E_u''')$ where
$V_u''' = V_u'' \cup \{z_i \mid 1 \le i \le m\}$,
$E_u''' = E_u'' \cup \{(z_i, z_{i+1}) \mid 1 \le i \le m-1\} \cup \{(u, z_1)\}$.

Let $G = (V, E)$ where
$V = V_v''' \cup V_u'''$,
$E = E_v''' \cup E_u''' \cup \{(u, v)\}$.

$\mathcal{E}[v] \ge B$ if and only if v is the optimal vertex in G. Since the size of V is less than $2q(1/\rho+1)|E_v|^2 + 2|E_v| + (31/\rho+1)|V_v| + 51/\rho$, the size of G is polynomial in the size of G_v. Therefore, OPTLOC is #P-hard since BEXPSZ is #P-hard.

4.3.2. Explanation and correctness. Clearly, $\mathcal{E}[v] \ge B$ if and only if $\mathcal{E}[v]+d \ge B+d$, for some rational term d, which we describe later. We, therefore, build a graph G_u' with optimal vertex u and $\mathcal{E}_{G_u'}[u] = B + d$. At the same time we form G_v' by augmenting G_v such that $\mathcal{E}_{G_v'}[v] = \mathcal{E}_{G_v}[v] + d$. We then augment both G_u' and G_v', forming G_u'' and G_v'', respectively, to ensure that either u or v or both are the optimal vertices in both G_u'' and G_v''. We then augment both G_u'' and G_v'', forming G_u''' and G_v''', respectively, to ensure that u and v do not have the same expected size. At this point, $\mathcal{E}_{G_v}[v] \ge B$ if and only if v is the optimal vertex in $(V_u''' \cup V_v''', E_u''' \cup E_v''' \cup \{(u, v)\})$.

Since the expected number of operational links from u to some u_i is $B'\rho$,

$$\mathcal{E}_{G_u}[u] = 1 + B'\rho.$$

Connecting a vertex to a "chain" of k vertices increases the expected size of the group containing that vertex by $\sum_{1\le j\le k} \rho^j$. We form G_v' from G_v by adding b_k chains of length $k-1$ for every $b_k \ge 1$. Therefore,

$$\mathcal{E}_{G_v'}[v] = \mathcal{E}_{G_v}[v] + \sum_{b_i \ge 1} \sum_{j=1}^{i-1} b_i \rho^j.$$

Likewise, we form G_u' from G_u by adding b_k chains of length k for every $b_k \ge 1$. Adding a chain of length k to G_u' and of length $k-1$ to G_v' produces a net increase of $b_k\rho^k$ in the difference between $\mathcal{E}_{G_u'}[u]$ and $\mathcal{E}_{G_v'}[v]$. Therefore,

$$\begin{aligned}\mathcal{E}_{G_u'}[u] &= 1 + B'\rho + \sum_{b_i \ge 1} \sum_{j=1}^{i} b_i \rho^j \\ &= 1 + B'\rho + \sum_{b_k \ge 1} b_i \rho^i + \sum_{b_i \ge 1} \sum_{j=1}^{i-1} b_i \rho^j \\ &= B + \sum_{b_i \ge 1} \sum_{j=1}^{i-1} b_i \rho^j.\end{aligned}$$

Now $\mathcal{E}_{G_v}[v] \ge B$ if and only if $\mathcal{E}_{G_v'}[v] \ge \mathcal{E}_{G_u'}[u]$. But OPTLOC tells which vertex is optimal in the entire graph, not which of u and v is better. Therefore, we must ensure that either u or v is the optimal vertex. Clearly, $\mathcal{E}_{G_v'}[v] \ge 1$ and $\mathcal{E}_{G_v'}[t] \le c$ for all $t \in V_v'$. By adding $\lceil (c+1)/\rho \rceil$ neighbors to v, we increase the expected group size of v by $\rho\lceil (c+1)/\rho \rceil$ and ensure, by 4.6, that v is the optimal vertex in G_v''. Likewise for u in G_u'. Therefore,

$$\mathcal{E}_{G''_v}[v] = \mathcal{E}_{G'_v}[v] + \rho\left\lceil (c+1)/\rho \right\rceil = \mathcal{E}_{G_v}[v] + \sum_{b_i \geq 1} \sum_{j=1}^{i-1} b_i \rho^j + \rho\left\lceil (c+1)/\rho \right\rceil,$$

$$\mathcal{E}_{G''_u}[u] = \mathcal{E}_{G'_u}[u] + \rho\left\lceil (c+1)/\rho \right\rceil = B + \sum_{b_i \geq 1} \sum_{j=1}^{i-1} b_i \rho^j + \rho\left\lceil (c+1)/\rho \right\rceil.$$

Now $\mathcal{E}_{G_v}[v] \geq B$ if and only if v is the optimal vertex in $(V''_u \cup V''_v, E''_u \cup E''_v)$, provided $\mathcal{E}_{G''_v}[v] \neq \mathcal{E}_{G''_u}[u]$. If, however, $\mathcal{E}_{G''_v}[v] = \mathcal{E}_{G''_u}[u]$, or equivalently $\mathcal{E}_{G_v}[v] = B$, we cannot be sure which of u and v will be called *optimal*, since OPTLOC, is indifferent in this case. Therefore, we introduce G'''_v and G'''_u such that

$$\begin{aligned}
\mathcal{E}_{G'''_v}[v] &= \mathcal{E}_{G''_v}[v] + \sum_{j=1}^{m+1} \rho^j \\
&= \mathcal{E}_{G_v}[v] + \sum_{b_i \geq 1} \sum_{j=1}^{i-1} b_i \rho^j + \rho\left\lceil (c+1)/\rho \right\rceil + \sum_{j=1}^{m} \rho^j + \rho^{m+1}, \\
\mathcal{E}_{G'''_u}[u] &= \mathcal{E}_{G''_u}[u] + \sum_{j=1}^{m} \rho^j \\
&= B + \sum_{b_i \geq 1} \sum_{j=1}^{i-1} b_i \rho^j + \rho\left\lceil (c+1)/\rho \right\rceil + \sum_{j=1}^{m} \rho^j.
\end{aligned}$$

Since B is a multiple of $1/q^m$, $\mathcal{E}_{G_v}[v] \geq B$ if and only if v is the optimal vertex in $(V''_u \cup V''_v, E''_u \cup E''_v)$.

We form G by connecting G'''_u and G'''_v with an edge from u to v. Clearly, "optimality" is preserved.

Therefore, $\mathcal{E}_{G_v}[v] \geq B$ iff v is the optimal vertex in G, and G can be achieved in time polynomial in the size of G_v. Since BEXPSZ is #P-hard, OPTLOC, is also #P-hard. □

4.4. Generalizing. Since probability graphs model a subset of the networks with arbitrary, nonuniform link reliabilities and networks with both fallible sites and fallible links, the #P-completeness result of the previous section applies to these more complex networks. Also, AboElFotoh and Colbourn have shown the #P-completeness of the CON problem where vertices, rather than edges, are subject to failure [1]. Using this result, the proof given in this paper can easily be modified to include radio broadcast networks and other networks modeled by graphs with fallible vertices and infallible edges. This also includes single, bus networks like Ethernet, where the link reliability can be factored out of the availability equation.

5. Concluding remarks. Although #P-complete in general, the determination of the optimal location for the resource is solvable for some systems. Since often a network for an existing database is built incrementally around the database, the current location may be optimal. As is shown in [2], [5], [12], [21], the single copy availability can be efficiently determined for regular network topologies, such as ring, single-bus, fully connected, and for series-parallel networks. Since, for these topologies, the single copy availability can be calculated in polynomial time by calculating the expected group size, $\mathcal{E}[v] = \sum_{u \in V} \mathcal{P}r\big(c(v, u)\big)$, for each site in V, the placement problem can be solved in polynomial time. It may also be possible to solve efficiently the placement problem for networks with fixed, deterministic routing algorithms, since the number of possible paths connecting two sites may not be a function of the size of the network, or the paths may be mutually independent.

Although calculating the expected group size is feasible in some special cases, it may be unnecessary to do so in real systems. Instead, each site could record the actual number of access requests submitted to sites within its group, and the site with the largest number can be made the location of the copy. (Requiring that a site record the number of access requests, rather than the number of sites, accommodates a nonuniform access request distribution.) This method maximizes availability because the number of access requests "seen," that is, those requests submitted within a site's group, is the same as the number of access requests that would be granted if the resource were located at that site, since communication is symmetric.

If the past network performance and the access request distribution are indicative of future behavior, this technique will lead to optimal copy placement. This method does not require a priori knowledge of the network topology, hardware reliability, or access request distribution, and it adjusts automatically to unanticipated changes in any of the these system parameters. These characteristics are precisely those necessary for any protocol that hopes to relocate an object as an access distribution change. Preliminary simulation work with this method has been encouraging [14].

In summary, we have analyzed a fundamental problem that seeks the optimal location for resources. Here optimality is obtained, not by minimizing a cost metric, but by maximizing availability, that is, the probability that an arbitrary access request is submitted to a site connected to the resource. We have shown that this optimal placement problem and two related network reliability problems are #P-complete and, therefore, likely to be computationally tractable only in very small networks. Given the increasing importance of distributed computing environments, the development of practical and efficient on-line approximation techniques is an important area for further research.

REFERENCES

[1] H. M. AboElFotoh and C. J. Colbourn, *Computing the two-terminal reliability for radio broadcast networks*, IEEE Trans. on Reliability, 1991, to appear.

[2] Daniel Barbara and Hector Garcia-Molina, *The reliability of voting mechanisms*, IEEE Trans. on Computers, C-36 (1987), pp. 1197–1208.

[3] R. G. Casey, *Allocation of copies of a file in distributed systems*, in Proceedings AFIPS 1972 SJCC, AFIPS Press, 1972, pp. 617–625.

[4] W. W. Chu, *Optimal file allocation in a multi-computer information system*, IEEE Trans. Comput., 18 (1969), pp. 885–889.

[5] Charles J. Colbourn, *The Combinatorics of Network Reliability*, Oxford University Press, London, 1987.

[6] S. A. Cook, *The complexity of theorem proving procedures*, in Proceedings of the Third ACM Symposium on Theory of Computing, ACM, May 1971, pp. 151–158.

[7] Lawrence W. Dowdy and Derrell V. Foster, *Comparative modes of the file assignment problem*, Computing Surveys, 14 (1982), pp. 287–313.

[8] K. P. Eswaran, *Placement of records in a file and file allocation in a computer network*, Inform. Proces., 74 (1974).

[9] Howard Frank and Ivan T. Frisch, *Communication, Transmission, and Transportation Networks*, Addison-Wesley, Reading, MA, 1971.

[10] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, New York, 1979.

[11] Bezalel Gavish and Olivia R. Liu Sheng, *Dynamic file migration in distributed computer systems*, Comm. Appl. Math. Comput., 33 (1990), pp. 177–189.

[12] E. N. Gilbert, *Random graphs*, Ann. of Math. Statistics, 30 (1959), pp. 1141–1144.

[13] M. Jerrum, *On the Complexity of Evaluating Multivariate Polynomials*, Ph.D. thesis, University of Edinburgh, 1981.

[14] Donald B. Johnson and Larry Raab, *Complexity of network reliability and optimal database placement problems*, Tech. report PCS-TR91-167, Dartmouth College, Hanover, NH, 1991.

[15] ———, *Effects of replication on data availability*, Internat. J. Comput. Simulation, 1 (1991), pp. 373–392.

[16] DONALD B. JOHNSON AND LARRY RAAB, *Effects of replication on the duration of failure in distributed databases*, Tech. report PCS-TR91-185, Dartmouth College, Hanover, NH, 1991.

[17] ———, *A tight upper bound on the benefits of replication and consistency control protocols*, in Proceedings of the 10th Symposium on Principles of Database Systems, ACM, May 1991, pp. 75–81.

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

[19] C. V. RAMAMOORTHY AND BENJAMIN W. WAH, *The isomorphism of simple file allocation*, ACM Trans. Comput. Systems, 32 (1983), pp. 221–232.

[20] ARNIE ROSENTHAL, *Computing the reliability of complex networks*, SIAM J. Appl. Math., 32 (1977), pp. 384–393.

[21] A. SATYANARAYANA AND R. K. WOOD, *A linear time algorithm for computing k-terminal reliability in series-parallel networks*, SIAM J. Comput., 14 (1985), pp. 818–832.

[22] SEINOSUKE TODA AND MITSUNORI OGIWARA, *Counting classes are at least as hard as the polynomial-time hierarchy*, SIAM J. Comput., 21 (1992), pp. 316–328.

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

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

[25] ROBERT S. WILKOV, *Analysis and design of reliable computer networks*, IEEE Trans. Commun., 20 (1972), pp. 660–678.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 520–552, June 1994

POLYNOMIAL ALGORITHMS FOR HAMILTONIAN CYCLE IN COCOMPARABILITY GRAPHS*

JITENDER S. DEOGUN† AND GEORGE STEINER‡

Abstract. Finding a Hamiltonian cycle in a graph is one of the classical NP-complete problems. Complexity of the Hamiltonian problem in permutation graphs has been a well-known open problem. In this paper the authors settle the complexity of the Hamiltonian problem in the more general class of cocomparability graphs. It is shown that the Hamiltonian cycle existence problem for cocomparability graphs is in P. A polynomial time algorithm for constructing a Hamiltonian path and cycle is also presented. The approach is based on exploiting the relationship between the Hamiltonian problem in a cocomparability graph and the bump number problem in a partial order corresponding to the transitive orientation of its complementary graph.

Key words. Hamiltonian path, Hamiltonian cycle, cocomparability graphs, partial order, bump number

AMS subject classifications. 05C85, 05C45, 06A06, 68R10, 68Q20

1. Introduction.

The *Hamiltonian cycle* problem is one of the classical NP-complete problems [6], [13] on graphs. The *Hamiltonian problem* remains NP-complete even on such special classes as planar 3-connected graphs [6], bipartite graphs [16], split graphs [7], edge graphs [2], planar bipartite graphs [11], circle graphs [3], and grid graphs [11]. Relatively few classes of graphs are known for which the problem has a polynomial time solution. These include 4-connected planar graphs [8] and interval graphs [14]. The *Hamiltonian cycle* has been a well-known open problem for permutation graphs for several years [12]. In this paper we solve the Hamiltonian Path and Hamiltonian Cycle problems for a more general class of graphs known as cocomparability graphs.

It is shown that the Hamiltonian cycle existence and construction problems for cocomparability graphs have a polynomial solution. In the process we identify necessary and sufficient conditions for the existence of a Hamiltonian path or cycle in a cocomparability graph. An $O(n^3)$ time complexity algorithm for constructing a Hamiltonian cycle is also presented. Our solution method is based on exploiting the strong connection between the Hamiltonian problems in a cocomparability graph and the bump number problem in a partial order corresponding to a transitive orientation of its complementary graph. Determining whether a cocomparability graph has a Hamiltonian cycle will be referred to as the *existence problem*, and the problem of finding a Hamiltonian cycle, when it exists, is referred to as the *construction problem.* The Hamiltonian path problem on cocomparability graphs is solved using the bump number algorithm in [4], and the same approach is used for the Hamiltonian completion problem, that is, the problem of finding the minimum number of edges that should be added to make a non-Hamiltonian cocomparability graph Hamiltonian [4]. Polynomial algorithms for finding the bump number of a partial order were recently reported [10], [17], here we use the results from [10].

The paper is organized as follows: §2 contains the preliminary definitions and the most important features of the bump number algorithm that are used in this paper. In §3 we show that the bump number problem is strongly related to the *Hamiltonian path* problem in cocomparability graphs. This is followed by technical results on *"critical"* elements in *"auxiliary"*

*Received by the editors June 10, 1992; accepted for publication (in revised form) December 10, 1992. This research was supported in part by the Office of Naval Research grant N0014-91-J-1693 and by the Natural Sciences and Engineering Research Council of Canada grant A1798.

†Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska 68588-0115.

‡Management Science and Information Systems Area, Faculty of Business, McMaster University, Hamilton, Ontario, Canada L8S 4M4.

partial orders. The critical elements play a crucial role in the solution of the existence problem for a cocomparability graph. Our main results are presented in §4, where we show that a cocomparability graph has a Hamiltonian cycle exactly if a transitive orientation of its complementary graph has no critical element. An $O(n^3)$ algorithm for solving the existence problem in cocomparability graphs is contained in §5. This is followed by a much more elaborate $O(n^3)$ algorithm, which solves the Hamiltonian cycle construction problem in a cocomparability graph.

2. Preliminary definitions and results. For the standard graph-theoretic and order-theoretic notions not mentioned here we refer the reader to [1] and [7]. Throughout the paper we consider only finite, simple, undirected graphs. Let $G = (V, E)$ be a graph with vertex set V and edge set E, and let $|V| = n$. For any $A \subseteq V$, the *subgraph* induced by A, $G[A]$ is the graph with vertex set A and those edges from E that have both their endpoints in A. For $G = (V, E)$, its *complementary graph* $G^c = (V, E^c)$ has vertex set V and edge set $E^c = V \times V - E$. A *path* in a graph is a set of vertices $\{x_1, x_2, x_3, \ldots, x_{k-1}, x_k\}$ such that $(x_i, x_{i+1}) \in E$, $1 \leq i \leq k-1$. A *cycle* in a graph is defined similarly except that in a cycle $x_1 = x_k$. We will also view paths or cycles as the collection of their edges, when this is more convenient. A path (cycle) is called *Hamiltonian* if its edges cover all the vertices in G exactly once. It is easy to show that the following monotonicity property holds.

PROPERTY 2.1. *If $G = (V, E)$ has no Hamiltonian path (cycle) and $G' = (V, E')$ with $E' \subseteq E$, then G' has no Hamiltonian path (cycle) either.*

A partial order is denoted by $P = (V, <_P)$, where V is the (finite) ground set of elements or vertices and $<_P$ is an order relation, i.e., an irreflexive, antisymmetric and transitive binary relation. The elements of the order relation $(a, b) \in <_P$ for $a, b \in V$ are written as $a <_P b$ with the usual interpretation.

We say that a is *covered* by b (denoted $a <\cdot\, b$) if $a <_P b$ and there is no $c \in V$ with $a <_P c <_P b$. Two elements $a, b \in V$ are *comparable* in P if either $a <_P b$ or $b <_P a$. Otherwise they are said to be *incomparable* and denoted by $a \,||_P\, b$. A graph $G = (V, E)$, whose edges are exactly the comparable pairs in a partial order P on V, is called a *comparability graph*. A graph G is a *cocomparability graph* if its complementary graph G^c has a transitive orientation, corresponding to the comparability relations in a partial order, denoted by P_G. A cocomparability graph G has as many partial orders P_G, as is the number of transitive orientations of G^c. However, a partial order P uniquely determines its comparability graph and, thus, its cocomparability graph denoted by $G(P)$ and $G^C(P)$, respectively.

Interval orders are the partial orders whose ground set is a family of closed intervals on the real line, with $[a, b] < [c, d]$ in the partial order if $b < c$ as numbers. The cocomparability graphs of interval orders are the *interval graphs. Permutation graphs* are exactly those graphs that are both comparability and cocomparability graphs [7]. The Hamiltonian problem for permutation graphs has been a well-known open problem [12]. In this hierarchy interval graphs represent the largest class for which the Hamiltonian cycle problem was known to be polynomially solvable [14] until now. In this paper, we extend the polynomial solvability of the Hamiltonian cycle problem to the class of cocomparability graphs, which properly includes the class of interval graphs as well as the class of permutation graphs. Thus, the class of cocomparability graphs is a significantly larger class of graphs for which the Hamiltonian problem can be solved. Cocomparability graphs also represent a large class over which many domination problems are known to be polynomially solvable [15].

A set of pairwise incomparable elements in a partial order is called an *antichain*. For $A, B \subseteq V$, disjoint subsets of V, $A <_P B$ means that $a <_P b$ for all $a \in A$, $b \in B$. A *linear order* is a partial order without incomparable elements. A *linear extension* of a partial order $P = (V, <_P)$ is a linear order $L = (V, <_L)$ that *extends* P, i.e., $a <_P b$ implies $a <_L b$

for all $a, b \in V$. We will write linear orders as $L = x_1x_2 \ldots x_n$, where the sequence from left to right defines the order relation $<_L$, i.e., $x_1 <_L x_2 <_L \ldots <_L x_n$. Two consecutive elements x_i, x_{i+1} of L are separated by a bump if $x_i <_P x_{i+1}$. The total number of bumps in L is denoted by $b(L, P)$. The *bump number* $b(P)$ of P is the minimum number of bumps in some linear extension, i.e.,

$$b(P) = \min\{b(L, P) \mid L \text{ is a linear extension of } P\}.$$

A linear extension L of P with $b(L, P) = b(P)$ is called *(bump-)optimal.* It was proved in [5], [9] that the bump number is a *comparability invariant*, i.e., partial orders with the same comparability graph have the same bump number too.

The bump number algorithm [10] is of the *primal-dual* type. This means that it constructs simultaneously a primal solution, a solution of the original minimization problem, and a dual solution, a solution of an associated maximization problem — the *dual* problem. The optimality of both primal and dual solutions is proven by showing that their objective values are equal. In our context a *primal solution* corresponds to a *linear extension L* of P, and its objective value is given by $b(L, P)$. The *dual solutions* correspond to *special* partial orders P^* *contained* in $P = (V, <_P)$. *Containment* is defined with respect to inclusion of the order relations, i.e., $P^* = (V, <_{P^*})$ is *contained* in $P = (V, <_P)$ (or, equivalently, P *contains* or *extends* P^*) if $a <_{P^*} b$ implies $a <_P b$ for all $a, b \in V$. We denote containment by $P^* \subseteq P$. It is easy to see that the bump number is nondecreasing with respect to containment, that is,

$$(1) \qquad P^* \subseteq P \quad \text{implies} \quad b(P^*) \leq b(P).$$

The *special* partial orders are defined as follows. For natural numbers $r \geq 1$ and $s \geq 0$, the class $\psi_{r,s}(V)$ consists of all partial orders $P = (V, <_P)$ for which there is a partition of V into $r + 1$ sets $U, A_1, \ldots, A_r$ such that

$$(2) \qquad |A_i| \geq 1, \quad i = 1, \ldots, r; \qquad |U| = s,$$

and

$$(3) \qquad a <_P b \quad \text{iff} \quad a \in A_i, b \in A_j, \text{ and } i < j.$$

In other words, every partial order in $\psi_{r,s}(V)$ is a parallel composition (disjoint sum) of s isolated elements and the series composition (ordinal sum) of the antichains $A_1, \ldots, A_r$. The partial orders from the classes $\psi_{r,s}(V)$ were called *generalized weak orders* in [10]. It can easily be seen that $b(P) = \max\{0, r - 1 - s\}$ for any $P \in \psi_{r,s}(V)$.

THEOREM 2.2 ([10]). *For any partial order P,*

$$\max\{b(P^*) | P^* \subseteq P, P^* \text{ a generalized weak order}\}$$
$$= \min\{b(L, P) | L \text{ a linear extension of } P\}.$$

Our algorithm for the Hamiltonian cycle problem on cocomparability graphs uses a slightly modified version of the bump number algorithm. The modified algorithm is presented in the appendix. The reader interested in more detail is referred to [10]. Here we describe only some salient features of the modified version of the bump number algorithm that are essential to our treatment of the Hamiltonian problem. The bump number algorithm is a greedy algorithm that traverses the given partial order $P = (V, <_P)$ in a "layered" fashion and avoids creating bumps as long as possible. Originally, P is given by a directed covering graph (Hasse diagram) with layers $H_0, H_1, \ldots, H_h$ in which each element is on the highest possible layer. Following this layered scheme the maximal elements are on the highest layer

H_h (where h is the height of P), and elements covered by some element in layer H_i are on layer $H_{i-1}, i = 1, \ldots, h$. The algorithm traverses the layers $H_0, H_1, \ldots, H_h$ one after the other in increasing order, starting with layer H_0. Since each H_i is an antichain of P, a bump can only occur at the transition between adjacent layers H_{i-1}, H_i. On leaving the current layer H_{i-1}, a bump is avoided if there are points a, b with

$$\text{(A.1)} \qquad a \in H_{i-1}, b \in H_j \quad \text{with } j \geq i, a \,||_P\, b, \text{ and } b \,||_P\, v \text{ for all } v \in H_i.$$

In this case, b can be placed right after a in a linear extension of P. A pair (a, b) fulfilling (A.1) is called a *transition* from H_{i-1} into H_j. If $j > i$, b is said to be *pulled down* from layer H_j in order to enable a non-bump transition to H_i, and the element b is called a *bump cutter*. Pulling down b from H_j will make H_j smaller for later steps of the algorithm. This is expressed by updating the layers accordingly. So throughout the algorithm $H'_0 = H_0, H'_1, \ldots, H'_h$ denote the current layers, where $H'_i \subseteq H_i$ denotes the set of elements that are still remaining on H_i, i.e., have not been pulled down.

The algorithm looks for transitions from the current layer H'_{i-1}, while observing three "greedy" choice rules. These rules define the layer from which b should be taken, and if there are several possible b's on that layer, specify which b to choose (or rather, not to choose). A transition $(a, b) \in H'_{i-1} \times H'_j$ is called *close* if H'_j is the closest layer above the current layer H'_{i-1} for which a transition from H'_{i-1} exists; if $j = i$, then we call the transition *short*. Then the choice rules are as follows:

Rule 1: Prefer close transitions (a, b) from H'_{i-1}.

Rule 2: Among several close transitions $(a_1, b_1), \ldots, (a_k, b_k)$ from H'_{i-1} to H'_j, with $j > i$, prefer (a_r, b_r) such that b_r is not on a close transition (b_r, c) from the layer H'_j.

Rule 3: Among several short transitions $(a_1, b_1), \ldots, (a_k, b_k) \in H'_{i-1} \times H'_i$ prefer (a_1, b_1) and (a_2, b_2) such that $b_1 \neq b_2$ and a_1 and a_2 are not the endpoints of chosen transitions to H'_{i-1}.

Rule 1 reflects that elements b' from higher layers with $a \,||_P\, b'$ might still be used for transitions (a', b') at later stages, while Rule 2 reflects that (b_r, c) might still be used when layer H_j is reached. Rule 3 tries to use *independent* pairs of short transitions, i.e., transitions whose endpoints are all different. It may be noted that this is a modification to the original algorithm in [10].

In going from layer H'_{i-1} to H'_i, the algorithm first looks for a transition $(a, b) \in H'_{i-1} \times H'_i$ (thus fulfilling Rule 1). If the element b is the same in all such transitions, the algorithm chooses an arbitrary such pair (a, b), and the element b is also called a bump cutter. Otherwise it reserves two transitions $(a, b), (a', b') \in H'_{i-1} \times H'_i$ with $b \neq b'$; if, in addition, a and a' can be chosen so that $a \neq a'$ and one or both are not endpoints of reserved transitions to H'_{i-1}, then they are selected this way (thus fulfilling Rule 3). The final choice between (a, b) and (a', b') is made later. If no transition $(a, b) \in H'_{i-1} \times H'_i$ exists, the algorithm looks for a transition $(a, b) \in H'_{i-1} \times H'_j$ with $j > i$ and, if there is one, chooses one according to Rules 1 and 2. Finally, if there is no transition (a, b) from H'_{i-1} to any $b \in H'_j, j \geq i$, the algorithm will create a bump in going from H'_{i-1} to H'_i.

When the choice of the transition is definite (i.e., all transitions $(a, b) \in H'_{i-1} \times H'_i$ have the same endpoint b or there is only one transition (a, b) into levels higher than H_i), then the linear extension to be constructed will take $H'_{i-1} - \{a\}$ in some permutation, then a followed by b, and then $H'_i - \{b\}$ in some permutation. The definite choice of a will then serve as the starting point for a backward procedure to make the final choice for the transitions between previous layers if the choice was delayed. In that case, the choice of a being the last element of H'_{i-1} determines the delayed choice of the transition from H'_{i-2} to H'_{i-1}. One of the two reserved transitions will not have a as second element, and this one is taken. Applying this principle recursively backward over all delayed decisions, the linear extension is constructed.

From now on, we will use the terminology used in the Algorithm bumpno (see the appendix), and whenever we refer to bump-optimal linear extensions, we will mean those "layered" extensions that could be generated by Algorithm bumpno. We will always use L to represent such a bump optimal linear extension. Define the intervals $L[x, u] = \{y| \, x \leq_L y \leq_L u\}$, $L| \, u = \{y| \, y \leq_L u\}$, and $u| \, L = \{y|u <_L y\}$. We will also use $G|u$ to denote the subgraph induced from G by the elements in $L|u$, for a given bump-optimal L of P_G. In the context of Hamiltonian cycles, we will use $L[x, u]$ to also denote the set of edges (v, w) if $v <\cdot_L w$ and $v, w \in L[x, u]$. The notation $HC(v, w)$ will be used to denote a Hamiltonian cycle that contains the edge (v, w), and if $v <\cdot_L w$, then v will always be the first endpoint shown for the edge (v, w).

3. Hamiltonian paths, cycles, and critical elements. The following result was proved in [4], but we include it here with a short proof, because this serves as a basis for the rest of the developments in the paper.

THEOREM 3.1. *A cocomparability graph $G = (V, E)$ has a Hamiltonian path if and only if $b(P_G) = 0$.*

Proof. If $b(P_G) = 0$, then the covering pairs in any bump-optimal linear extension of P_G form a Hamiltonian path in G. If $b(P_G) > 0$, then, by Theorem 2.2, P_G contains a generalized weak order $P_{r,s}$ with $b(P_G) = r - 1 - s$. Furthermore, it can easily be seen that, since $r - 1 - s > 0$, $G^C(P_{r,s})$ has no Hamiltonian path. Let E' be the edge set of $G^C(P_{r,s})$; then $E \subseteq E'$ and, by the monotonicity Property 2.1, G cannot have a Hamiltonian path. □

Based on the above proof, Theorem 3.1 also implies the following duality result:

COROLLARY 3.2. *A cocomparability graph $G = (V, E)$ has no Hamiltonian path if and only if there exists a generalized weak order $P_{r,s}$ on V such that $G^C(P_{r,s})$ has no Hamiltonian path and E is contained in E', the edge set of $G^C(P_{r,s})$.*

Naturally, since the existence of a Hamiltonian cycle also implies the existence of a Hamiltonian path, $b(P_G) = 0$ is a necessary condition for a cocomparability graph G to have a Hamiltonian cycle. This in itself is not sufficient, however, and our aim in the remainder of this section is to identify additional, necessary, and sufficient conditions.

DEFINITION 3.3. *An element $x \in V$ is called* critical *if $b(P_G - x) = b(P_G) + 1$, where $P_G - x$ is the subposet of P_G induced by $V - \{x\}$.*

LEMMA 3.4. *If $b(P_G) = 0$ and P_G has a critical element x, then G has no Hamiltonian cycle.*

Proof. Since x is critical, $b(P_G - x) = 1$. By Theorem 2.2 $P_G - x$ contains a generalized weak order $P_{r,s}$ with $r - 1 - s = 1$. Let $P_{r,s+1}$ be the disjoint union of $P_{r,s}$ and x (i.e., adding back x without any comparability relations). It is clear that $G^C(P_{r,s+1})$ is a graph with no Hamiltonian cycle.

However, G is contained in $G^C(P_{r,s+1})$; therefore, by the monotonicity Property 2.1 G has no Hamiltonian cycle. □

This lemma gives the first indication that the existence or nonexistence of critical elements plays a crucial role in determining whether a cocomparability graph has a Hamiltonian cycle. Naturally, the elements that are not on the entry or exit transition to or from a layer can always be removed from a layered optimal linear extension, without creating bumps, so they can never be critical.

DEFINITION 3.5. *An element $x \in H'_j$ is called* potentially critical *if x is the entry or exit point to or from H'_j in every optimal linear extension.*

LEMMA 3.6. *If $|H'_j| > 2$ and $x \in H'_j$ is the entry point to H'_j in every optimal linear extension, then x is critical.*

Proof. Suppose x is not critical; then there exists a linear extension L' of $P_G - x$ with $b(P_G - x) = b(L') = b(P_G)$. Let $a, b \in H'_j$ be the first and second elements, respectively, from H'_j in L'. Now, we want to insert x in L' between a and b without increasing the bump number. However, there may exist a $v \in H'_{j-1}$ such that $v < x$ in P_G but incomparable with every other element of H'_j and thus could appear after b in L'. But it is easy to see that such a v can be moved before a in L' without adding bumps. So we can assume that every element of H'_{j-1} is before a in L'. Thus x can be inserted between a and b in L'. This again adds no bumps and results in an optimal linear extension of P_G in which x is *not* the entry point into H'_j. □

The situation is somewhat more complicated if all optimal linear extensions *exit* a layer H'_j via the same point x, as it is indicated in Fig. 1. It is easy to see that every optimal linear extension of the P_G in Fig. 1 will exit H'_1 via x, but x is not critical because the linear extension $L' = badcefg$ of $P_G - x$ has no bumps. The point $e \in H_0$, which is below x but incomparable to every other point of H_1, moved up to become the exit point from $H_0 \cup (H_1 - x)$ in L'.

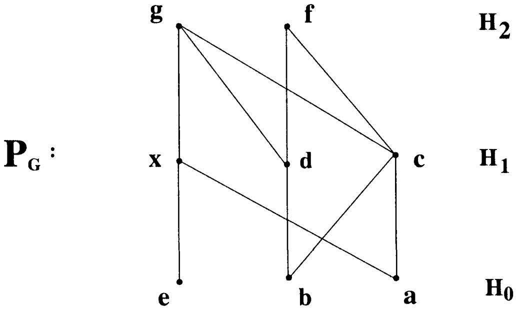


FIG. 1.

LEMMA 3.7. *Suppose that* $|H'_j| > 1$ *and* $x \in H'_j$ *is the only exit point for transitions from* H'_j; *then either* x *is critical or in every optimal linear extension of* $P_G - x$ *the exit point from* $H'_{j-1} \cup H'_j - x$ *is an* $e \in H'_{j-1}$ *such that* $e < x$ *and* $e \not< v$ *for* $v \in H'_j - x$ *in* P_G.

Proof. If x is not critical, then there is a linear extension L' of $P_G - x$ with $b(L') = b(P_G - x) = b(P_G)$. If L' exited $H'_{j-1} \cup H'_j - x$ via a $d \in (H'_j - x)$, then the subposet induced by the points in $L|x$ must have an optimal linear extension L_d; ending in d, we could replace its initial segment (up to d) by L_d, resulting in an optimal linear extension of P_G in which H'_j does not exit via x, a contradiction. Therefore, L' must exit $H'_{j-1} \cup H'_j - x$ via an $e \in H'_{j-1}$, which moved up to the jth layer after deleting x. □

If a layer H'_i, $|H'_i| \geq 2$, has two points x, y such that every short transition from H'_{i-1} to H'_i ends in x or y and every short transition from H'_i to H'_{i+1} starts in x or y, then x and y are also potentially critical, since in every optimal linear extension both x and y will have to be an entry or exit point (with these roles possibly interchanged in different linear extensions).

If a layer H'_i has two points x, y that are endpoints of short transitions from H'_{i-1} and two points v, w that are starting points of short transitions to H'_{i+1} with $|\{x, y, v, w\}| > 2$, then we can always construct an optimal linear extension L in which any designated point of H'_i can be made a nonentry and nonexit point, e.g., second from H'_i in L. Thus, a layer with this property cannot contain potentially critical points.

LEMMA 3.8. (1) *If* H'_i $(0 < i < h)$ *is not the bottom layer of a closed set,* $|H'_i| > 2$, *and it has exactly two points* x, y *that are the only endpoints for transitions from* H'_{i-1} *to* H'_i *as well as the only starting points for transitions between* H'_i *and* H'_{i+1}, *then* x *and* y *are critical.*

(2) *Let $|H_i'| \geq 2$, and let $x \in H_i$ be the only starting point of transitions from H_i'. If there is no $e \in H_{i-1}'$ that could move up to this layer when x is deleted and that is also the starting point of a transition to some H_j' $(j > i)$, then x is critical.*

Proof. (1) Since $|H_i'| > 2$, there exists a $c \in H_i' - \{x, y\}$ and c is above every element in H_{i-1}'. So if we delete x, the only entry point for a transition into H_i' is y. But then we have no transition from H_i' to H_{i+1}'; therefore, $b(P_G - x) > b(P_G)$. The proof is similar for $b(P_G - y)$.

(2) If we delete x from P_G, there is no element left that could serve as the starting point of a transition from H_i to H_{i+1}'; therefore, $b(P_G - x) > b(P_G)$. □

Consider a generalized weak order $P_{r,s}$ with $r - 1 - s \geq 0$ and its set of isolated elements $U \neq \emptyset$. Since $r - 1 - s \geq 0$, every element of U is used as a bump cutter in a bump-optimal linear extension of $P_{r,s}$. Therefore, every $u \in U$ must be critical. This observation yields:

LEMMA 3.9. *If $b(P_G) = 0$ and P_G contains a generalized weak order $P_{r,s}$ such that $r - 1 - s = 0$ and $s > 0$, then P_G has a critical element and G has no Hamiltonian cycle.*

The lemma means that the existence of a Hamiltonian cycle in G implies not only $b(P_G) = 0$ but also that the bump number algorithm must produce a dual order for P_G that is simply an antichain on V. This can happen if either the bump number algorithm uses the procedure `close` only once at the end (i.e., the dual order is built only by the successive applications of the procedure `grow`) or `close` is used more than once but all antichains in the dual order get eventually combined into a single antichain by the procedure `collapse`. In the following we deal with these two cases separately.

4. Hamiltonian cycles.

4.1. Strictly grown orders. In the first case the dual order is *strictly grown*, i.e., the bump number algorithm has always found at least two transitions into the next layer with different endpoints.

Case 1*a*. In this subcase the starting points of the transitions are also different, i.e., we have two independent transitions between each pair of consecutive layers, and these independent transitions cover at least 3 points on every layer, except the top and bottom ones. In this situation we can easily construct a Hamiltonian cycle in G using this set R of reserved transitions. First we find two node-disjoint paths between the top and bottom layers of P_G by the following procedure.

```
procedure twopaths($P_G$; $R$)
    1. Start by choosing one of the reserved transitions from $H_0'$ to $H_1'$.
    2. Suppose $(a,b) \in H_i' \times H_{i+1}'$ was the last transition chosen. If $H_{i+1}'$ is the top layer, then the selected transitions and edges form a path $\pi_1$ and go to 3. If $b$ is the starting point of a reserved transition $(b,c)$ to the next layer, $H_{i+2}'$, then choose $(b,c)$ and repeat Step 2; otherwise choose a reserved transition $(d,e) \in H_{i+1}' \times H_{i+2}'$ with the restriction that $d$ is not the end point of a reserved transition from $H_i'$ to $H_{i+1}'$. Connect $(a,b)$ and $(d,e)$ adding the horizontal edge $(b,d) \in H_{i+1}' \times H_{i+1}'$ from $G$. Repeat Step 2.
    3. The remaining reserved transitions (after adding some $(u,v) \in H_i' \times H_i'$ type horizontal edges from $G$, if necessary) form a path $\pi_2$ from the top layer to $H_0$.
```

The fact that this procedure will construct two node-disjoint paths between the top and bottom layers follows from the observation that after each execution of Step 2 there will be exactly one reserved transition left to and from each intermediate layer and none of these cover points that are covered by transitions in π_1. Furthermore, since each intermediate layer had at least 3 points on the reserved transitions, at least one of π_1 or π_2 must contain a horizontal edge $(u, v) \in H_i' \times H_i'$ for each intermediate layer H_i'. This means that any points $w \in H_i'$, missed by both π_1 and π_2, can be inserted between these u and v. Finally the Hamiltonian cycle in G can be completed by connecting the end of π_1 to the start of π_2 in H_h' and the end of π_2 to the start of π_1 in H_0' and by inserting any missed points from H_0 or H_h by horizontal edges in the corresponding layers.

The entire construction is illustrated in an example in Fig. 2. Suppose the transition $(1, 4) \in \pi_1$ was chosen. Since the node 4 is also the start of a transition, (4, 7) is added to π_1. After this, (6, 9) and (8, 11) are the next available reserved transitions, but (8, 11) cannot be chosen since 8 is the end point of the transition (3, 8); thus, (6, 9) is added to π_1 with (7, 6) joining it to the earlier built part. In the next iteration (9, 13) must be chosen since 9 is both the end and start of a reserved transition. In the last iteration through Step 2 both transitions (12, 17) and (16, 19) are available, so choose, say (16, 19), and join it to π_1 by (13, 16). This completes $\pi_1 = (1, 4, 7, 6, 9, 13, 16, 19)$. The remaining reserved transitions along with the horizontal edges (12, 15), (10, 11), and (3, 5) form $\pi_2 = (17, 12, 15, 10, 11, 8, 3, 5, 2)$. The only point missed from an intermediate layer is 14. Since both π_1 and π_2 have a horizontal edge from the layer of 14, we insert 14, say, between 13 and 16, in π_1. Finally, π_1 and π_2 are combined into the Hamiltonian cycle HC = (1, 4, 7, 6, 9, 13, 14, 16, 19, 18, 20, 17, 12, 15, 10, 11, 8, 3, 5, 2, 1), after including the points 18 and 20, missed from H_h'.

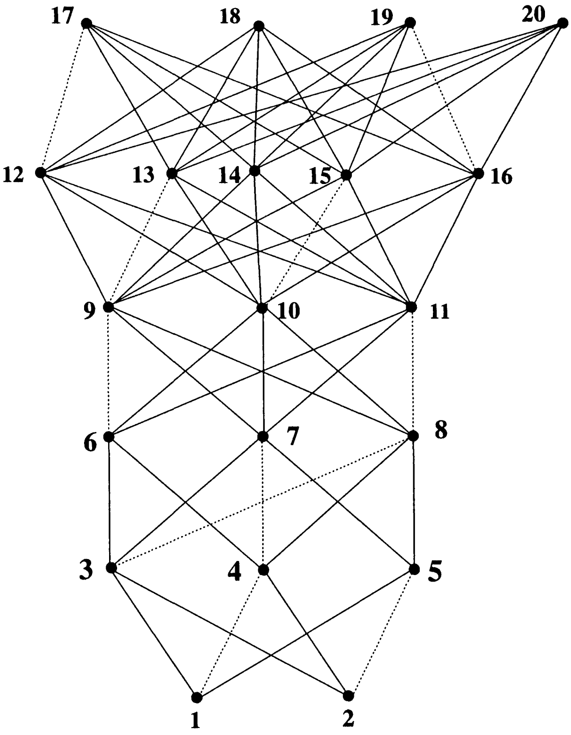


FIG. 2. *P_G with reserved transitions represented by dotted lines.*

Remark: We note that since missed points from H_0' or H_h' can be inserted in any order at the end, we can assume without loss of generality that if $a, b \in H_h'$ (or H_0'), such that at most one of them is on a reserved transition, then HC(G) will contain the edge (a, b). For example, the HC(G) constructed in the example contains the horizontal edges (19, 18), (18, 20), and (20, 17), but by changing the order of insertions it could be made to go through either one of the edges (19, 20) or (18, 17). The only horizontal edge it is forced to avoid is (17, 19), the edge connecting the endpoints of the reserved transitions. This is an important property of the Hamiltonian cycle constructed, which will be used later.

Case 1*b*. In this subcase there are two independent transitions between each pair of consecutive layers, but there is an intermediate layer H_i' where all transitions end and start in the same two points x and y. Applying the procedure `twopaths` would still yield two node-disjoint paths π_1 and π_2, the same way as in Case 1a. If $|H_i'| = 2$, then we can again use the same method as before to construct a Hamiltonian cycle, but if H_i' contains a third point c, then both π_1 and π_2 would miss c and neither one of them has a horizontal edge in the layer H_i', where c could be inserted later. This situation is illustrated in Fig. 3.

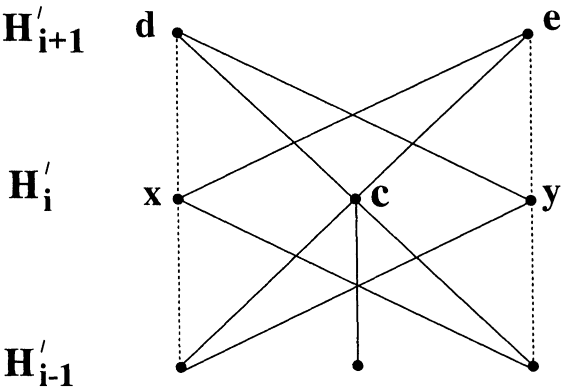


FIG. 3. *A Type* 1 *blocker.*

LEMMA 4.1. *Let G be a cocomparability graph. Suppose that $b(P_G) = 0$ and P is strictly grown. Let H_i' with $|H_i'| > 2$ be an intermediate layer that contains two points x and y such that there are transitions both ending and starting only in the points x and y, implying that every other point $c \in H_i'$ is above every element in H_{i-1}' and below every element in H_{i+1}'. If P_G^{i+1} denotes the restriction of P_G to $H_0' \cup H_1' \cup \cdots \cup H_{i+1}'$, then x and y are critical in P_G^{i+1}.*

Proof. The lemma is a direct consequence of Lemma 3.8(1) applied to P_G^{i+1}. □

We will refer to a layer H_i that satisfies the conditions of Lemma 4.1 as a *Type* 1 *blocker*. Of course, the existence of such a blocker does not necessarily preclude the existence of a Hamiltonian cycle in G, as it may be possible to pull down some other element, from a layer above H_{i+1}, that could play the role of a bump cutter between H_i and H_{i+1} in $P_G - x$ or $P_G - y$. Therefore, the elements x and y of Lemma 4.1 are *potentially critical*, as defined earlier, but whether these are critical or not really depends on the rest of P_G.

Case 1*c*. In this subcase there is a layer H_i, with $|H_i'| \geq 2$ and element $x \in H_i'$ such that all short transitions from H_i' to H_{i+1}' start in x. This situation is depicted in Fig. 4. In this case the procedure `twopaths` could not find two node-disjoint paths between H_0' and H_h'. We will refer to the layer H_i' with these properties as a *Type* 2 *blocker*. The element x of a Type 2 blocker may or may not be critical (cf. Lemma 3.7 and Lemma 3.8(2)), as it may be possible to pull down some other element, from a layer above H_{i+1}, to act as a bump cutter between H_i and H_{i+1} in $P_G - x$. In another situation, as shown in the second example of

Fig. 4, an element $e \in H'_{i-1}$, that can move up "to act" as a bump cutter between H'_i and H'_{i+1} in $P_G - x$ may exist. Therefore, the element x is only potentially critical.

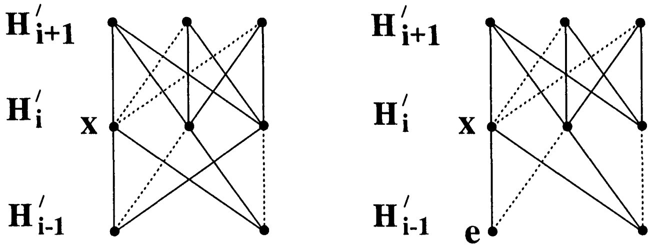


FIG. 4. *Type* 2 *blockers.*

We will use HP(u, v) to denote a Hamiltonian path between vertices u and v in a graph to be specified. Similarly HC(u, v) is used to denote a Hamiltonian cycle that includes edge (u, v). The following technical lemma will be used repeatedly.

LEMMA 4.2. *Let G_1 and G_2 be two graphs having exactly two common vertices u and v. Let $G = G_1 \cup G_2$ be the graph whose vertex set and edge set is the union of those in G_1 and G_2. If* $\mathrm{HP}_1(u, v)$ *and* $\mathrm{HP}_2(u, v)$ *are Hamiltonian paths in G_1 and G_2, respectively, then* $\mathrm{HC} = \mathrm{HP}_1(u, v) \cup \mathrm{HP}_2(u, v)$ *is a Hamiltonian cycle in G. Similarly, if* $\mathrm{HC}_1(u, v)$ *and* $\mathrm{HC}_2(u, v)$ *are Hamiltonian cycles in G_1 and G_2 respectively, then* $\mathrm{HC} = \mathrm{HC}_1(u, v) \cup \mathrm{HC}_2(u, v) - \{(u, v)\}$ *is a Hamiltonian cycle in G. Furthermore,* $\mathrm{HP}_1(u, v) \cup \mathrm{HC}_2(u, v) - \{(u, v)\}$ *and* $\mathrm{HC}_1(u, v) - \{(u, v)\} \cup \mathrm{HP}_2(u, v)$ *are also Hamiltonian cycles in G.*

Proof. Obvious. □

THEOREM 4.3. *Let G be a cocomparability graph. Suppose that $b(P_G) = 0$ and P_G is strictly grown. G has a Hamiltonian cycle if and only if P_G has no critical element.*

Proof. In one direction the theorem follows from Lemma 3.4. For the other direction, assume that P_G has no critical element. We use induction on the number of blockers. If P_G has no blockers, then we have Case 1a and a Hamiltonian cycle can be constructed as described earlier. In correspondence with our remark at the end of Case 1a, we make our inductive hypothesis slightly stronger than what is stated in the theorem:

Inductive hypothesis. Suppose $b(P_G) = 0$, the dual order is strictly grown, and P_G has no critical element and at most m blockers. If $c, x \in H'_0$ such that at most one of them is on a reserved transition, then there is a Hamiltonian cycle HC(c, x) in G.

Let us consider now a P_G satisfying the assumptions of the theorem. Further assume that P_G has no critical element and $m + 1$ blockers. Let H'_i be the lowest level blocker in P_G .

Case i. *The lowest level blocker H'_i is of Type* 1 *with potentially critical elements x and y.* Without loss of generality (wlog) let L be an optimal linear extension of P_G in which x is the last element from H'_i. Since x is not critical, there is a linear extension L' of $P_G - x$ such that $b(P_G - x) = b(L') = 0$. Every $v \in H'_i - \{x, y\}$ is above every element of H'_{i-1} and below every element of H'_{i+1}, so L' must use y as its entry point to H'_i. Therefore, there must exist a $u \in H'_j\ (j > i + 1)$ that acts as a bump cutter between H'_i and H'_{i+1} in L', and this vertex u can be used to bypass H'_i in our Hamiltonian cycle. First we note that the bypass element u must be incomparable to every point occurring between x and u in L, i.e., $v \not< u$ in P_G for every $v \in H'_k\ (i < k \leq j)$. Let $c \in H'_i - \{x, y\}$ be the last element from H'_i in L'. We can assume that x covers c in L. The subposet induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_i$ has no blockers and satisfies the earlier discussed Case 1a, thus there is a Hamiltonian cycle $\mathrm{HC}_1(c, x)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_i$ from G. Furthermore, $\mathrm{HC}_2(c, x) = \{(c, x), L[x, u], (u, c)\}$ is also a cycle and, by Lemma 4.2, $\mathrm{HC}_3 = \mathrm{HC}_1(c, x) \cup HC_2(c, x) - \{(c, x)\}$ is a Hamiltonian cycle in the subgraph $G|u$.

If H'_j is not a blocker, then we can always rearrange L (and HC_3) so that u is the second element from H'_j in L. (We will call this a *rearrangement of Type* 1). If a denotes the element covered by u in this rearranged L, then HC_3 contains the edge (a, u). The subgraph induced by $H'_j \cup H'_{j+1} \cup \cdots \cup H'_h$ satisfies the assumptions of the inductive hypothesis with fewer than $m + 1$ blockers and u is not a potentially critical element (H'_j has none), so there is a $\mathrm{HC}_4(a, u)$ in this subgraph. By Lemma 4.2 again, $\mathrm{HC} = \mathrm{HC}_3(a, u) \cup HC_4(a, u) - \{(a, u)\}$ is a Hamiltonian cycle in G.

If H'_j is a Type 2 blocker and u is its potentially critical element, then we cannot have an $e \in H'_{j-1}$ (with $e < u$ in P_G) that could move up to the jth level in $P_G - u$ to act as exit point from this level in the sense of Lemma 3.7 because of our observation that the bypass element u is incomparable to every point between x and itself. Therefore, since u is not critical, there must be an $a \in H'_j - u$ that is the starting point of a transition from H'_j to some $v \in H'_k$ ($k > j + 1$). If a is the only such starting point and, in addition, $|H'_j| > 2$, then a and u cannot be the only endpoints for short transitions between H'_{j-1} and H'_j because this would make them critical by Lemma 3.8(1). So irrespective of the size of H'_j (even if $|H'_j| = 2$) we can assume that L (and HC_3) can be rearranged so that a is covered by u in L. (We will call this a *rearrangement of Type* 2). Therefore, (a, u) will be an edge in the rearranged HC_3. On the other hand, the subgraph induced by $\{a, u\} \cup H'_{j+1} \cup \cdots \cup H'_h$ satisfies the assumptions of the inductive hypothesis with at most m blockers, so there is an $\mathrm{HC}_4(a, u)$ in it. By Lemma 4.2, $\mathrm{HC} = \mathrm{HC}_3(a, u) \cup \mathrm{HC}_4(a, u) - \{(a, u)\}$ is a Hamiltonian cycle in G.

If H'_j is a Type 2 blocker but u is not its potentially critical element, then let x' be its potentially critical element and three subcases exist. In the first subcase $|H'_j| > 2$ and there is an $a \in H'_j - x'$ that acts as exit point from H'_j in the bump-free linear extension L' of $P_G - x'$. It may be noted that the existence of L' is guaranteed since x' is not critical. Since $|H'_j| > 2$, if $a \neq u$, then we can assume that u is not the entry point to H'_j in L by Lemma 3.6 and we can rearrange L so that u covers a in L (a rearrangement of Type 2). If u is the only possible exit point from $H'_j - x'$ (i.e., $a = u$) in L', then x' and u are the only exit points from H'_j and by Lemma 3.8(1) there must be an $a \in H'_j - \{x', u\}$ that can be the entry point to H'_j in L. Rearrange L and HC_3 so that a is covered by u in L. In either case this leads to an HC_3 that contains the edge (a, u). Let B denote the set of points following u in L. The subposet induced by $\{a, u\} \cup B$ satisfies the assumptions of the inductive hypothesis with at most m blockers. Therefore, there is an $\mathrm{HC}_4(a, u)$ in its cocomparability graph. Thus $\mathrm{HC} = \mathrm{HC}_3(a, u) \cup \mathrm{HC}_4(a, u) - \{(a, u)\}$ is a Hamiltonian cycle in G. In the second subcase $|H'_j| = 2$ ($H'_j = \{x', u\}$) and u acts as the exit point from H'_j in the bump-free linear extension L' of $P_G - x'$. The subposet $\{u, x'\} \cup B$, where $B = u \mid L$, satisfies the assumptions of the inductive hypothesis with at most m blockers, so there is an $\mathrm{HC}_4(u, x')$ in its cocomparability graph. By rearranging the segment $L|x'$ we can make u the last point from H'_j. Using this rearranged segment for rearranging HC_3 we can make $(x', u) \in \mathrm{HC}_3$. Then $\mathrm{HC} = \mathrm{HC}_3 \cup \mathrm{HC}_4(u, x') - \{(u, x')\}$ is a Hamiltonian cycle in G. In the third subcase, irrespective of the size of H'_j, an $e \in H'_{j-1}$ acts as the exit point from the jth layer in the bump-free linear extension of $P_G - x'$, in the sense of Lemma 3.7. Since e must be incomparable to every $v \in H'_j - x'$, we can rearrange L so that e is covered by u in L. (We call this a *rearrangement of Type* 3). The resulting HC_3 will have (e, u) as an edge. Let $B = u|L$. If we delete the comparability relation (e, x') in the subposet $\{e, u\} \cup B$, then it will satisfy the assumptions of the inductive hypothesis with at most m blockers, so there is an $\mathrm{HC}_4(e, u)$ in its comparability graph. (It is clear that x' and e will be the endpoints of reserved transitions used in HC_4, so e and x' will not be neighbors in HC_4). Therefore, $\mathrm{HC} = \mathrm{HC}_3(e, u) \cup \mathrm{HC}_4(e, u) - \{(e, u)\}$ is a Hamiltonian cycle in G.

If H'_j is a Type 1 blocker, then we can assume wolg that u is the exit point from H'_j in L. Accordingly, HC_3 is a Hamiltonian cycle on the subgraph $G|u$ induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_j$. Since H'_j is a Type 1 blocker, we can repeat the argument used for bypassing H'_i — since u is not critical, there is a $u' \in H'_k$ $(k > j+1)$ that acts as bump cutter between H'_j and H'_{j+1} in a bump-free linear extension L'' of $P_G - u$. If H'_k is not a blocker or it is a Type 2 blocker, then we have reduced this case to one of those discussed earlier. On the other hand, if H'_k is a Type 1 blocker again, then we can repeat the bypassing argument for it too. After a finite number of repetitions we either must cover the whole graph G by the Hamiltonian cycle or end up using a bump cutter either from a nonblocker layer or from a Type 2 blocker.

Case ii. *The lowest level blocker H'_i is of Type* 2, *with potentially critical element x.* Let L be an optimal linear extension of P_G. Since x is not critical, there is a linear extension L' of $P_G - x$ with $b(P_G - x) = b(L') = 0$. Let $u \in H'_j$ $(j \geq i+1)$ be the first (the bypass) element in L' from above H'_i, and let c be the element covered by u in L'.

If $c \in H'_i$ and $H'_i = \{x, c\}$, then there is a $\mathrm{HC}_1(c, x)$ in the subgraph of G induced by $L|x$, since this subgraph has no blocker. If $c \in H'_i$ and $|H'_i| > 2$, then every $d \in H'_i - \{c, x\}$ is before c in L', by definition, so there is a transition from H'_{i-1} to H'_i with some endpoint $d \in H'_i - \{c, x\}$. Using this as one of the reserved transitions in the application of `twopaths` to the poset induced by $H'_0 \cup H'_1 \cup \cdots \cup (H'_i - x)$ means that at most one of c and x may lie on another reserved transition. Therefore, by our remark at the end of Case 1a, there is a $\mathrm{HC}_1(c, x)$ in the subgraph G^i, induced by $L|x$.

If $c \in H'_{i-1}$ then c is not comparable to any element in $H'_i - x$. Since every $d \in H'_i - x$ is before c in L', by definition, there is a $c' \in (H'_{i-1} - c)$ and $d \in (H'_i - x)$ such that (c', d) is a short transition between H'_{i-1} and H'_i. If c and c' are not the only endpoints of short transitions between H'_{i-2} and H'_{i-1} or $H'_{i-1} = H'_0$ or $|H'_{i-1}| = 2$, then, by our remark at the end of Case 1a, there is a $\mathrm{HC}_0(c, c')$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1}$. If $|H'_{i-1}| > 2$, $i - 1 > 0$, and every $v \in H'_{i-1} - \{c, c'\}$ is above every element of H'_{i-2}, then one of these v must be the starting point of a short transition into H'_i, since otherwise the layer H'_{i-1} would be a Type 1 blocker (contradicting that H'_i is the lowest level blocker, see Fig. 5), and this v can play the role of c' in the previous sentence. In summary, there must exist an $\mathrm{HC}_0(c, c')$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1}$, such that c' is the starting point of a short transition (c', d) to H'_i and c is not comparable to any element in $H'_i - x$. $\mathrm{HC}_0(c, c')$ can be enlarged into an $\mathrm{HP}_1(c, x)$ of $G|x$ by inserting d, followed by $H'_i - \{x, d\}$ in any order, and finally x between c' and c. We will refer to this sequence of operations as a *rearrangement of Type* 4.

If $c \in H'_i$, then it can be assumed that c is covered by x in L by Lemma 3.6 (c cannot be the only entry point to H'_i if $|H'_i| > 2$). Therefore, $\mathrm{HC}_2(c, x) = \{(c, x)\} \cup L[x, u] \cup \{(u, c)\}$ is a cycle and $\mathrm{HC}_3 = \mathrm{HC}_1(c, x) \cup$ $\mathrm{HC}_2(c, x) - \{(c, x)\}$ is a Hamiltonian cycle in $G|u$. If $c \in H'_{i-1}$, then $\mathrm{HP}_2(c, x) = L[x, u] \cup \{(u, c)\}$ is a path and $\mathrm{HC}_3 = \mathrm{HP}_1(c, x) \cup HP_2(c, x)$ is a Hamiltonian cycle in $G|u$. In either one of the previous situations HC_3 can be enlarged into a Hamiltonian cycle in G, by a construction identical to the one used in Case i; therefore, we do not repeat it here. □

We also note that we proved a bit more than the existence of a Hamiltonian cycle in G. Because of its importance in the remainder of the paper, we state this stronger result as a separate corollary:

COROLLARY 4.4. *Suppose $b(P_G) = 0$, P_G is strictly grown, and P_G has no critical element. If $x \in H'_i$ is the exit point from H'_i in the optimal linear extension L, $c \in H'_i$ is not potentially critical, and there is a transition (c, u) from H'_i to $u \in H'_j$ $(j > i)$, then there is a Hamiltonian cycle $HC_1(c, x)$ in the subgraph of G induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_i$ and there is a $HC_2(c, x)$ in the subgraph of G induced by $\{c, x\} \cup H'_{i+1} \cup \cdots \cup H'_h$.*

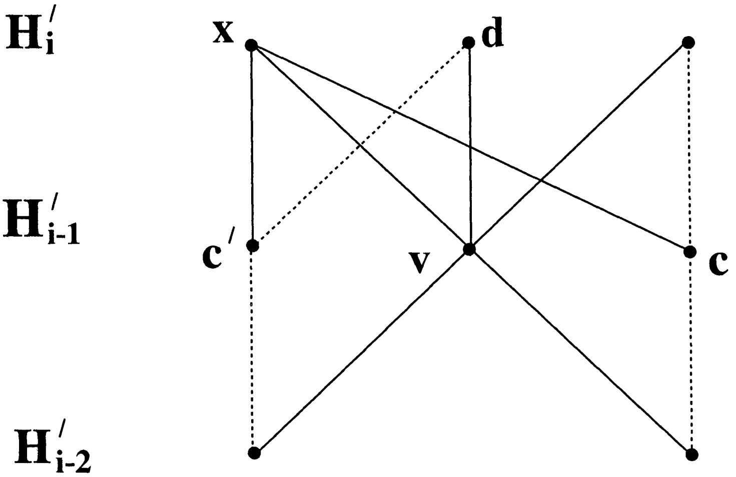


FIG. 5.

As we have seen, even though a strictly grown partial order may contain blockers, when the potentially critical elements in a blocker are not critical, we can include in a Hamiltonian cycle the elements of such a blocker. We will refer to this as the blocker being *bypassed* and will call the element u the *bypass element* for the blocker.

In Fig. 6 we show a P_G satisfying the conditions of Theorem 4.3. In this poset H_2 is a Type 1 blocker and H_4, H_6, H_{10} are Type 2 blockers. Suppose bumpno has found for P_G the bump-free linear extension L = 1, 2, 3, ..., 37, 38, 39. The first blocker is H_2, so applying twopaths to $H_0 \cup H_1 \cup H_2$ we get HC$_1$ = (2, 1, 5, 6, 7, 8, 4, 3, 2). The element $x = 8$ is not critical because we have the transition (7, 15) with $u_1 = 15$ as a bypass element for H_2. Using $L[8, 15]$, we have HC$_2$(7, 8) = (7, 8, 9, 10, 11, 12, 13, 14, 15, 7). Since $(7, 8) \in$ HC$_1$, no rearrangement is necessary, and we get HC$_3$ = HC$_1 \cup$ HC$_2$(7, 8) $-$ {(7, 8)} = (2, 1, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8, 4, 3, 2). The bypass element $u_1 = 15$ is the potentially critical element of the Type 2 blocker H_4. The transition (14, 19) could be used to exit from $H_4 - \{15\}$ in $P_G - \{15\}$. Therefore, the subgraph G_1 and subposet induced by {14, 15, 16, 17, 18, ..., 39} satisfy the conditions of Theorem 4.3 (if the original P_G satisfied them), with L_1 = 14, 15, 16, 17, ..., 38, where L_1 is the same as L after 17. We should find a Hamiltonian cycle, containing the edge (14, 15), on this subgraph. The point 15 is potentially critical in the bottom layer {14, 15} of this subposet, but the transition (14, 19) bypasses it. This gives HC$_4$ (14, 15) = (15, 16, 17, 18, 19, 14, 15), with bypass element u_2 = 19. This u_2 is the entry point in L to the Type 2 blocker H_6, but it is not the only possible entry to H_6, so it does not have to be critical. We can rearrange L_1 so that $u_2 = 19$ becomes the second point in it from H_6. The new L_1 = 14, 15, 16, 18, 17, 20, 19, 21, ..., 38, which follows the old L_1 after 21. The resulting rearrangement of HC$_4$ yields a new HC$_4$(14, 15) = (15, 16, 18, 17, 20, 19, 14, 15), which now contains the additional point 20. To bypass the Type 2 blocker H_6, we can use the bypass element $u_3 = 25$, which is the endpoint of the transition (19, 25) between $H_6 - \{21\}$ and H_7. The subgraph G_2 and corresponding subposet induced by $H_6 \cup H_7 \cup \cdots \cup H_{13}$ satisfy the conditions of Theorem 4.3 if G_1 satisfied them, with L_2 = 20, 19, 21, ..., 38, where L_2 follows the pattern in L_1 after 21. We should find a Hamiltonian cycle, containing the edge (20, 19), in G_2. Building our cycle in G_2 to the bypass element $u_3 = 25$ yields HC$_5$ (20, 19) = (19, 20, 21, 22, 23, 24, 25, 19). The bypass element $u_3 = 25$ is the entry point to H_8. We can make u_3 the second point in L_2 by rearranging L_2 to 20, 19, 21, 22, 23, 24, 27, 25, 26, 29, 28, 30, ..., 38, where it follows the old L_2 after 30, and the new HC$_5$ (20, 19) = (19, 20,

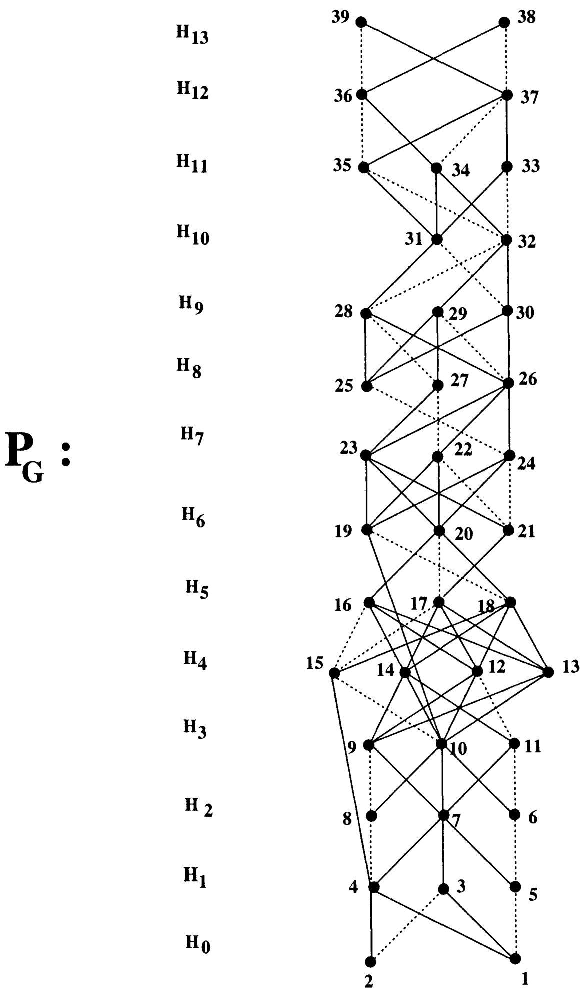


FIG. 6. *An example of the construction in strictly grown orders.*

21, 22, 23, 24, 27, 25, 19). The subgraph G_3 and subposet induced by $H_8 \cup H_9 \cup \ldots \cup H_{13}$ satisfy the conditions of Theorem 4.3, if G_2 satisfied them, with $L_3 = 27, 25, 26, 29, 28, 30, \ldots, 38$, where L_3 follows the pattern in L_2 after 30. We should find a Hamiltonian cycle, containing the edge (25, 27), in G_3. Building our cycle in G_3 yields HC_6 (25, 27) = (25, 27, 28, 32, 31, 30, 29, 26, 25). We reached the Type 2 blocker H_{10}, but, this time, there is no transition starting in H_{10} to bypass its potentially critical point 32. On the other hand, the element $e = 29$ could move up to become the exit point from $H_{10} - \{32\}$. We can rearrange L_3 so that 32 is covering e. The new $L_3 = 26, 25, 27, 28, 30, 31, 29, 32, \ldots$, with no change after 32, and HC_6 becomes HP_6 (29, 32) = (29, 26, 25, 27, 28, 30, 31, 32). By deleting the comparability relationship $29 < 32$, 29 moves up to H_{10}. The subgraph G_4 and subposet induced by $\{29, 32\} \cup H_{11} \cup \cdots \cup H_{13}$ satisfy the conditions of Theorem 4.3 if G_3 satisfied them, with $L_4 = 29, 32, 33, \ldots, 38$, where L_4 follows the pattern of L_3 after 33. We should find a Hamiltonian path from 29 to 32 in G_4. Building our path in G_4 yields HP_7 (29, 32) = (29, 33, 34, 37, 38, 39, 36, 35, 32). All that is left is to combine the Hamiltonian cycles and paths generated into a cycle in G. Let

$$\begin{aligned}
\mathrm{HC}_8 &= \mathrm{HP}_6 \cup \mathrm{HP}_7 &&= (29, 33, 34, 37, 38, 39, 36, 35, 32,\\
&&&\quad 31, 30, 28, 27, 25, 26, 29);\\
\mathrm{HC}_9 &= \mathrm{HC}_8 \cup \mathrm{HC}_5 - \{(25, 27)\} &&= (29, 33, 34, 37, 38, 39, 36, 35, 32,\\
&&&\quad 31, 30, 28, 27, 24, 23, 22, 21, 20,\\
&&&\quad 19, 25, 26, 29);\\
\mathrm{HC}_{10} &= \mathrm{HC}_9 \cup \mathrm{HC}_4 - \{(20, 19)\} &&= (29, 33, 34, 37, 38, 39, 36, 35, 32,\\
&&&\quad 31, 30, 28, 27, 24, 23, 22, 21, 20,\\
&&&\quad 17, 18, 16, 15, 14, 19, 25, 26, 29);\\
\mathrm{HC} &= \mathrm{HC}_{10} \cup \mathrm{HC}_3 - \{(15, 14)\} &&= (29, 33, 34, 37, 38, 39, 36, 35, 32,\\
&&&\quad 31, 30, 28, 27, 24, 23, 22, 21, 20,\\
&&&\quad 17, 18, 16, 15, 7, 6, 5, 1, 2, 3, 4,\\
&&&\quad 8, 9, 10, 11, 12, 13, 14, 19, 25, 26, 29);
\end{aligned}$$

HC is a Hamiltonian cycle in G.

4.2. Collapsed dual orders. As we have seen before, certain (bypass) elements can be used to grow a Hamiltonian cycle beyond a blocker in strictly grown orders. The situation is quite similar for bump cutters: Suppose we have grown our Hamiltonian cycle $\mathrm{HC}_1(c_1, x)$ to include the top layer of closed set A_k, with x being the exit point in L from H_i', the top layer of A_k. If the bump cutter u_k is not critical, then there is a bump-free linear extension L' of $P_G - u_k$. (In the following discussion an indexed u_k always refers to the (kth) bump cutter between A_{k-1} and A_k). This L' must use a $u \in u_k|L$ as a bypass element; i.e., if u is the first element from $u_k|L$ in L', then u must be the endpoint of a transition (c, u) for some $c \in H_i'$, where c is the exit point from H_i' in L'. If we can rearrange L and $\mathrm{HC}_1(c_1, x)$ so that c moves into the position immediately before x, i.e., we get $\mathrm{HC}_1(c, x)$, then $\mathrm{HC}_u = \mathrm{HC}_1(c, x) \cup L[x, u] \cup \{(u, c)\} - \{(c, x)\}$ is a Hamiltonian cycle in $G|u$, that is, HC_1 was enlarged to include u_k and points beyond u_k .

The next lemma states some important properties for bypass elements.

LEMMA 4.5. *Suppose H_i' is either a blocker or the top layer of a closed set A_k and x is the exit point from H_i' in L. If u is the bypass element for the blocker H_i' or the bump cutter u_k, then*:

(a) *If u is used to bypass the bump cutter u_k, then u is incomparable to every point that is between u_k and itself in L; if u is used to bypass the blocker H_i', then u is incomparable to every point that is between x and itself in L.*

(b) *If $u \in H_j'$, then $|H_j'| \geq 2$ and H_j' is not the bottom layer of a closed set.*

(c) *If $u \in H_j'$, then H_j' cannot be a Type 1 blocker if there is a $v \in H_{j-1}'$ with $v < u$ in P_G.*

Proof. Part (a) follows from the fact that u is the endpoint of a transition between H_i' and H_{i+1}'.

A layer H_j' can have one point only if it is the bottom layer of a closed set (otherwise the bumpno algorithm would have made its single element a bump cutter). If H_j' were the bottom layer of a closed set, then for every $w \in H_j'$ and $v \in H_{j-1}'$ we have $v < w$ in P_G. Since $j - 1 \geq i$, by (a) this shows that the bypass element u could not have come from a bottom layer H_j'.

Suppose H_j' is a Type 1 blocker and there is a $v \in H_{j-1}'$ with $v < u$ in P_G. This is in contradiction with (a) again, proving (c). □

The next two lemmas discuss rearrangements that involve the bypass element.

LEMMA 4.6. *Suppose we have constructed an* $\mathrm{HC}_1(c_1, x)$ *on* $G|x$, *where* x *is the exit point in* L *from a layer* H_i' *that was either a blocker or the top layer of a closed set* A_k; $\mathrm{HC}_1(c_1, x)$ *was enlarged into an* $\mathrm{HC}_u = \mathrm{HC}_1(c_1, x) \cup \mathrm{HC}_2(c_1, x) - \{(c_1, x)\}$, *where* $\mathrm{HC}_2(c_1, x) = \{(c_1, x), L[x, u], (u, c_1)\}$ *with a bump cutter* $u = u_j \in U$ $(j > k)$ *the bypass element* (*for the blocker* H_i' *or the bump cutter* u_k). *Let* H_t' $(t > i)$ *be the top layer of the closed set* A_j, *and let* d *be the point covered by* u_j *in* L *and* c *be the entry point to* H_t' *in* L. *Then* L *and* HC_u *can be rearranged so that* c *is the exit from* H_t' *in* L, u *covers* c *in* L *and* (c, u) *is an edge in* HC_u.

Proof. By Lemma 4.5, u must be incomparable to every point in H_t', in particular, to c. From the bump number algorithm we know that L can be rearranged, so that c becomes its exit point from H_t', without affecting its optimality. How does this rearrangement affect HC_u? If H_t' is also the bottom layer of the closed set A_j, then the bump cutter u_{j-1} is incomparable to every element of H_t', so we can simply exchange c and d in L and HC_u. Otherwise, if H_t' was entered via the short transition (e_1, c), with $e_1 \in H_{t-1}'$, then making c the exit point from H_t' forces us to use a different entry point, but there must exist another short transition (e_2, v_1) between H_{t-1}' and H_t' to a $v_1 \in H_t' - c$, since H_t' was reached by a `grow` step by Lemma 4.5(b). If $e_3 \neq e_2$ was the entry point to H_{t-1}' in L, then we can easily make e_2 the exit point from H_{t-1}' and rearrange L and HC_u between e_3 and u in the following order: e_3, $H_{t-1}' - \{e_3, e_2\}$ in any order, followed by $e_2, v_1, H_t' - \{v_1, c\}$ in any order, and followed by c. If e_2 was the entry point to H_{t-1}' in L, then making it the exit from H_{t-1}' forces us to use a different entry point. However, following an argument similar to the one above, there is a $v_2 \in H_{t-1}' - e_2$ such that there is a transition (e_3, v_2) between H_{t-2}' and H_{t-1}', and so on. Inductively carrying on with this, we either reach a layer where we are not forced to make further changes or reach the blocker layer H_i' for which u was used as the bypass element. (If we reach H_i' by these forcings, then all layers between H_i' and H_t' must either be Type 1 blockers or be of size 2.) If H_i' was a Type 2 blocker, with x its potentially critical (exit) element, then the forced changing of the entry point to H_{i+1}' to a, say, $e_r \in H_{i+1}'$, does not force further changes since x must be the starting point for the transition ending in e_r. Suppose now that H_i' was a Type 1 blocker itself, with potentially critical elements x and y, with x being the exit from H_i' in L. If $\mathrm{HC}_1(c_1, x)$ reached x as a bypass element, then, by Lemma 4.5(c), x must be incomparable (in P_G) to every $v \in H_{i-1}'$; therefore, H_{i-1}' cannot be a blocker; if $\mathrm{HC}_1(c_1, x)$ reaches x by an application of the algorithm `twopaths`, then H_{i-1}' cannot be a blocker because, to bypass it, HC_1 would have had to reach at least u. Let $e_r, e_{r+1} \in H_{i-1}' (e_r \neq e_{r+1})$ be the starting points of the transitions (e_r, y) and (e_{r+1}, x). Forcing y to be the exit point from H_i' forces us to change L, to make it use (e_{r+1}, x) instead of (e_r, y) between H_{i-1}' and H_i'. This, however, forces no further changes, since either H_{i-1}' is the bottom layer of A_j or by `rules 1` and `2` of `bumpno` L must have used a transition (a, b) from H_{i-2}' to H_{i-1}', with $b \in H_{i-1}' - \{e_r, e_{r+1}\}$. In any case the set of reserved transitions used by `twopaths` does not change, we only have to change $\mathrm{HC}_1(c_1, x)$ into an $\mathrm{HC}_1(c_1, y)$ by moving c_1 to become a neighbor of y instead of a neighbor of x (by the proper selection of horizontal edges in H_i'). This $\mathrm{HC}_1(c_1, y)$ can be continued by combining it with a new $\mathrm{HC}_2(c_1, y) = \{(c_1, y), L[y, u], (u, c_1)\}$, using the rearranged L. In this L the point c in the statement of the lemma is the exit point from H_t', and (c, u) is an edge as required. □

LEMMA 4.7. *Suppose we have constructed an* $\mathrm{HC}_1(c_1, x)$ *on* $G|x$, *where* x *is the exit point in* L *from a layer* H_i' *that was either a blocker or the top layer of a closed set* A_k; $\mathrm{HC}_1(c_1, x)$ *was enlarged into an* $\mathrm{HC}_u = \mathrm{HC}_1(c_1, x) \cup \mathrm{HC}_2(c_1, x) - \{(c_1, x)\}$, *where* $\mathrm{HC}_2(c_1, x) = \{(c_1, x), L[x, u], (u, c_1)\}$ *with* $u \in H_j'$ *the bypass element* (*for the blocker* H_i' *or the bump cutter* u_k). *If* H_j' *is a Type* 2 *blocker and* u *is not its potentially critical* (*exit*) *point, then* $|H_j'| > 2$, *and* L *and* HC_u *can be rearranged so that* u *becomes the second point from* H_j' *in* L.

Proof. Suppose that, contrary to the lemma, $H'_j = \{x', u\}$ with x' its potentially critical (exit) element. If H'_j was entered in L via a transition (e_1, u) from some $e_1 \in H'_{j-1}$, then $e_1 < x'$ in P_G, since otherwise we should have $e_1 \in H'_j$ by the definition of the layers. On the other hand, there is an $e_2 \in H'_{j-1}$ with (e_2, x') a transition, since H'_j was reached by a `grow` step by Lemma 4.5(b). If $x <_L e_2$, then (e_2, u) is also a transition, by Lemma 4.5(b), which would make $e_2 \in H'_j$. If $x \not<_L e_2$, then $i = j-1$, and H'_i cannot be the top layer of a closed set or a Type 1 blocker, it can be only a Type 2 blocker. In this case, however, (x, x') and (x, u) would be transitions from H'_{j-1} to H'_j, contradicting that $x \in H'_{j-1}$. So $|H'_j| > 2$ and $x' \neq u$ is its exit point. If u is not the entry point to H'_j in L, then u can be freely moved to be second from H'_j in L. If u was the entry point, then, by Lemma 3.6, there must be an alternative optimal linear extension in which u is not the entry point to H'_j. This means that there is a $d \in H'_j - \{x', u\}$ and $c \in H'_{j-1}$ with (c, d) a transition. We can rearrange L so that it uses d for entry to H'_j and u follows d in L. This may force backward changes in the transitions used earlier, but it can be proved, by using essentially the same argument as in the proof of the previous lemma, that the 'chain' of forcings would have to end at or before reaching H'_i. This means that these changes could affect only the HC_2 of the lemma but not HC_1. Accordingly, the rearrangement of L can easily be incorporated into HC_u. □

In this section we deal with orders that were *not* strictly grown, but we state our main theorem so that it includes the strictly grown case too:

THEOREM 4.8. *Let G be a cocomparability graph. Then G has a Hamiltonian cycle if and only if $b(P_G) = 0$ and P_G has no critical element.*

Proof. Theorem 3.1 and Lemma 3.4 prove that both conditions are necessary. To prove that they are also sufficient we use induction on the number of closed sets created by the bump number algorithm. If $m = 1$ (i.e., dual order is strictly grown), then Theorem 4.3 proves sufficiency.

Inductive hypothesis: If $b(P_G) = 0$, P_G has no critical element, and its dual order has at most m closed sets, then G has a Hamiltonian cycle. Furthermore, if $c, x \in H_0$ are not both potentially critical, then G has a Hamiltonian cycle that contains the edge (c, x).

Consider now a P_G that satisfies the conditions of the theorem and whose dual order has $m+1$ closed sets, $A_1, A_2, \ldots, A_{m+1}$ with bump cutters $u_1, u_2, \ldots, u_m$ between them.

Case 1. There is a blocker in A_1 that cannot be bypassed by using an element from A_1, i.e., the blocker contains an element that is critical in the subposet induced by A_1. (Note that a blocker can never be the top layer of a closed set, by definition.) Let $H'_i \subseteq A_1$ be the one of these blockers that is on the lowest level, and let $x \in H'_i$ be the last element from H'_i in an optimal linear extension L of P_G. Since x is not critical, there is a linear extension L' of $P_G - x$ such that $b(L') = 0$. Let c be the last element in L' from $H'_{i-1} \cup H'_i$ and u be the (bypass) element immediately following it. If $u \in A_1$ were true, then this u could be used to bypass the blocker H'_i within A_1, so $u \in V - A_1$, i.e., there is no transition from $H'_i - x$ to any layer of A_1 above H'_i.

Case 1a. H'_i is a Type 1 blocker.

By Corollary 4.4 there is a $\mathrm{HC}_1(c, x)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1} \cup H'_i$. On the other hand, $\mathrm{HC}_2(c, x) = \{(c, x), L[x, u], (u, c)\}$ is a cycle too, so $\mathrm{HC}_u = \mathrm{HC}_1(c, x) \cup \mathrm{HC}_2(c, x) - \{(c, x)\}$ is a Hamiltonian cycle in $G|u$, bypassing the blocker H'_i.

Case 1b. H'_i is a Type 2 blocker.

If $c \in H'_i$, then by Corollary 4.4 there is a $\mathrm{HC}_1(c, x)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_i$. $\mathrm{HC}_1(c, x)$ can be enlarged into a Hamiltonian cycle HC_u of $G|u$ the same way as in Case 1a.

If $c \in H'_{i-1}$, then c is not comparable to any element in $H'_i - x$. Since every $v \in H'_i - x$ is before c in L', by definition, there is a $c' \in (H'_{i-1} - c)$ and $d \in (H'_i - x)$ such that (c', d)

is a short transition between H'_{i-1} and H'_i. If c and c' are not the only endpoints of short transitions from H'_{i-2} to H'_{i-1}, or $H'_{i-1} = H_0$, or $|H'_{i-1}| = 2$, then, by Corollary 4.4, there is a $\mathrm{HC}_0(c', c)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1}$. If $|H'_{i-1}| > 2$, $i - 1 > 0$, and every $v \in H'_{i-1} - \{c, c'\}$ is above every element of H'_{i-2}, then one of these v must be the starting point of a short transition into H'_i, since otherwise the layer H'_{i-1} would be a Type 1 blocker that could not be bypassed within A_1, contradicting the assumption that H'_i is on the lowest level of these blockers. So one of the $v \in H'_{i-1} - \{c, c'\}$ could play the role of c' above. In summary, there is a $\mathrm{HC}_0(c', c)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1}$ such that c' is the starting point of a short transition (c', d) to H'_i and c is not comparable to any element in $H'_i - x$. $\mathrm{HC}_0(c', c)$ can be enlarged into a $HP_1(c, x)$ of $G|x$ by inserting d, $H'_i - \{x, d\}$, and x between c' and c. $\mathrm{HP}_1(c, x)$ can be enlarged into a Hamiltonian cycle HC_u of $G|u$ the same way as in Case 1a.

Case 2. There is no blocker that cannot be bypassed within A_1. Let H'_i be the top layer of A_1 and $x \in H'_i$ the last element from H'_i in an optimal linear extension L of P_G. Since u_1, the bump cutter between A_1 and A_2, is not critical, there is a linear extension L' of $P_G - u_1$ with $b(L') = 0$. Let $c \in H'_i$ be the last element in L' from H'_i and $u \in V - \{A_1 \cup u_1\}$ be the element immediately following it.

Case 2a. If $x \neq c$, then, by Corollary 4.4, there is a $\mathrm{HC}_1(c, x)$ in the subgraph induced by A_1. $\mathrm{HC}_2(c, x) = \{(c, x)\} \cup L[x, u] \cup \{(u, c)\}$ is also a cycle, so $\mathrm{HC}_u = \mathrm{HC}_1(c, x) \cup HC_2(c, x) - \{(c, x)\}$ is a Hamiltonian cycle in the subgraph $G|u$.

Case 2b. If x and c cannot be made different, i.e., for all possible choices of L and L' they always exit H'_i through the same $x = c$, then by Lemma 3.7 there must be an $e \in H'_{i-1'}$, which would move up to become the exit point from $H'_{i-1} \cup H'_i - x$ if x was deleted from P_G. It also follows that there is a, possibly new, bypass element $u \in V - A_1$, with (e, u) a transition and u immediately following e in an optimal linear extension L'' of $P_G - x$. We show that e cannot be potentially critical: Since e is the exit point from $H'_{i-1} \cup H'_i - x$ in L'', e is incomparable to every $v \in H'_i - x$ and there must be at least one transition from $H'_{i-1} - e$ to $H'_i - x$, say (f, f'). So H'_{i-1} cannot be a Type 2 blocker. Suppose H'_{i-1} is a Type 1 blocker with potentially critical elements e and f. By Lemma 3.8(1) e and f would have to be critical in the poset induced by A_1, contradicting the assumption that every blocker within A_1 can be bypassed within A_1. So e is not a potentially critical element (and H'_{i-1} is not a blocker), and therefore, there is an optimal linear extension L of P_G in which e is the second last element from H'_{i-1} with a $g \in H'_{i-1}$ the exit point. By Corollary 4.4 there is a $\mathrm{HC}_1(e, g)$ in the subgraph induced by $H'_0 \cup H'_1 \cup \cdots \cup H'_{i-1}$. $\mathrm{HC}_2(e, g) = \{(e, g), L[g, u], (e, u)\}$ is also a cycle, and $\mathrm{HC}_u = \mathrm{HC}_1(e, g) \cup \mathrm{HC}_2(e, g) - \{e, g\}$ is a Hamiltonian cycle in the subgraph $G|u$.

Combining all the above cases, we have shown the existence of a Hamiltonian cycle HC_u in the subgraph $G|u$. We also note that the bypass element u was always chosen by the linear extension L', and therefore, it must be the closest (lowest level) bypass element available. In the remainder of the proof we show how to enlarge this HC_u into a Hamiltonian cycle of G using the inductive hypothesis. Let B denote the set of elements following u in L.

Case i. The element u in HC_u is one of the bump cutters, say, $u = u_j$.

Let H'_t be the top layer in A_j and $y \in H'_t$ be the element immediately preceding u_j in L. Since u_j is not critical, there is a linear extension L' of $P_G - u_j$ with $b(L') = 0$. Let $c \in H'_t$ be the last element in L' from A_j.

If $u_j = u_1$ (this can happen only if HC_u was constructed earlier under Case 1 of the proof), then there must be a transition from a $v \in A_1$ to a (lowest level) $w \in A_k (k > 1)$, since otherwise the dual order of P_G could not have been collapsed. Since (v, w) is not a short transition, there must also exist a transition from the top layer of A_1 to w (otherwise transitivity would preclude (v, w) being a transition), so we assume, without the loss of generality, that $v \in H'_t$. Furthermore, since $u_j = u_1$ was the bypass element for a blocker, u_1 must be incomparable to every element of H'_t, in particular, to v. We can assume, without the loss of generality, that the optimal linear extension L, which we used to construct HC_u, had v as its last element from H'_t. Therefore, (v, u_1) was an edge in HC_u. $\mathrm{HC}_2(v, u_1) = \{(v, u_1), L[u, w], (w, v)\}$ is a cycle, and $\mathrm{HC}_w = \mathrm{HC}_u \cup \mathrm{HC}_2(v, u_1) - \{(v, u_1)\}$ is a Hamiltonian cycle in the subgraph $G|w$. If $w = u_j$ for some j, then we have $j > 1$. If $w \in H'_j$ for some j, then we have a cycle that reached into a layer above H'_t and this situation will be dealt with under Case ii.

So if our HC_u goes to a bump cutter $u = u_j$, we can now assume that $j > 1$. If $y = c$, then the subposet induced by $(y, u_j) \cup B$ has no critical element, bump number 0, and fewer than $m + 1$ closed sets, so by the inductive hypothesis there is a $\mathrm{HC}(y, u_j)$ in the subgraph induced by these elements. By Lemma 4.2 $\mathrm{HC} = \mathrm{HC}_u \cup \mathrm{HC}(y, u_j) - \{(y, u_j)\}$ is a Hamiltonian cycle in G.

If $y \neq c$ and c was the entry point to H'_t in L, then, by Lemma 4.6, L and HC_u can be rearranged so that c becomes the exit point from H'_t in L and we have the previous case again. If $y \neq c$ and c was not the entry point to H'_t in L, then c can be freely moved in L so that c becomes the exit point from H'_t in L ($u = u_j$ is not comparable to any element of H'_t since u was a bypass element for something below the layer H'_t, so $c \parallel u_j$ in P_G). Thus, we have the case of the previous paragraph again.

Case ii. The bypass element u is not one of $\{u_1, u_2, \ldots, u_m\}$ and $u \in H'_j$.

By Lemma 4.5(b) $|H'_j| \geq 2$ and we can repeat the argument used to continue the cycle HC_3 in Case i of the proof of Theorem 4.3. The only change needed is replacing the inductive hypothesis with the current one and using the argument to continue the current HC_u instead of HC_3 of the proof of Theorem 4.3. Therefore, we omit the details. □

Since there are very few known necessary and sufficient conditions for a graph to have a Hamiltonian cycle, we feel it is important to relate Theorem 4.8 in graph-theoretic terms:

COROLLARY 4.9. *A cocomparability graph $G = (V, E)$ has a Hamiltonian cycle if and only if*

(1) *G has a Hamiltonian path*;

(2) *For every $v \in V$ the subgraph $G - v$ induced by $V - \{v\}$ has a Hamiltonian path.*

Proof. The corollary is a direct consequence of Theorems 3.1 and 4.8. □

5. The algorithms. Theorem 4.8 means that we can easily state an algorithm for the *Hamiltonian Cycle Decision* problem on cocomparability graphs.

ALGORITHM 5.1.

```
procedure hamiltonian_cycle ($G$)
    Input:    A cocomparability graph $G$.
    Output:   YES if $G$ has a Hamiltonian cycle, NO otherwise.
    Step 1.   Find a transitive orientation of $G^C$ and
              call it $P_G$.
    Step 2.   Call bumpno($P_G$)
              If $b(P_G) > 0$ or ($b(P_G) = 0$ but the dual order is not
                  collapsed into a single antichain) then output
                  ''NO'' and STOP.
    Step 3.   For each potentially critical element $x \in P_G$ do;
                  Call bumpno($P_G - x$)
                  If $b(P_G - x) > 0$ then output ''NO'' and STOP.
              end;
    Step 4.   Output ''YES'' and STOP.
```

THEOREM 5.2. *Algorithm* 5.1 *solves the Hamiltonian Cycle Decision problem for a cocomparability graph G in $O(hn^2)$ time and $O(n^2)$ space, where n is the number of vertices in G and h is the height of P_G, that is, the size of a largest independent set of G.*

Proof. The correctness of the algorithm is a direct consequence of Theorem 3.1, Lemma 3.4, and Theorem 4.8.

The claimed complexity follows from the following observations: G^C and a transitive orientation for it can be found in $O(n^2)$ time and space [18]. The bumpno algorithm for P_G requires at most $O(n^2)$ time and space [10]. It is clear that P_G has at most h blockers or bump cutters, so it can have at most $O(h)$ potentially critical elements. Calling bumpno for all $P_G - x$, where x is potentially critical, will require $O(hn^2)$ time and $O(n^2)$ space. □

Algorithm 5.1 solves the *Hamiltonian Cycle Decision* problem but does not find such a cycle in case the answer is YES. In the remainder of this section we present an algorithm that also constructs a Hamiltonian cycle, if it exists.

Algorithm 5.3 starts with preprocessing steps, which identify certain cases when G cannot have a Hamiltonian cycle. After this the algorithm calls the main procedure hamiltonian, which takes its detailed (layered) input from bumpno and starts the construction of a Hamiltonian cycle layer by layer, by the previously defined procedure twopaths. The procedure twopaths builds two parallel paths through the layers in G until it either reaches the top layer H_h' or a blocker layer H_i' or the bump cutter u_1. In the first case the two independent paths can easily be joined into a Hamiltonian cycle in G. In the second case the two independent paths are joined into a Hamiltonian cycle covering the subgraph induced by $H_0' \cup H_1' \cup \cdots \cup H_i'$ and the procedure bypass1 or bypass2 is called depending on whether H_i' is a Type 1 or Type 2 blocker. In the third case the two independent paths are joined into a Hamiltonian cycle of the subgraph induced by the first closed set A_1 and the procedure bypass3 is called.

The procedure bypass1 (H_i'; x, y, $\mathrm{HC}_1(c_1, x)$) looks for a bypass element u that can be used to grow the current cycle HC_1 beyond the layer H_i'. If such u does not exist, then the potentially critical elements x, y in H_i' are critical and G has no Hamiltonian cycle. If a bypass element is found, then HC_1 is enlarged into an HC_u, covering all the points in $G|u$.

The procedure bypass2 (H_i'; x, $\mathrm{HC}_1(c_1, x)$) looks for a bypass element u that can be used to grow the current cycle HC_1 beyond the Type 2 blocker H_i'. If no u is found, then the potentially critical (exit) point x from H_i' is critical and G has no Hamiltonian cycle. If a bypass element is found, then HC_1 is enlarged into an HC_u, covering $G|u$.

The procedure bypass3 (H_i'; x, u_k, $\mathrm{HC}_1(c, x)$) looks for a bypass element u that can be used to grow the current cycle beyond the closed set A_k (i.e., bypass the bump cutter u_k). If such u does not exist, then the bump cutter u_k is critical and G has no Hamiltonian cycle. If a bypass element is found, then HC_1 is enlarged into an HC_u, covering $G|u$.

All three bypass procedures return the control to the main procedure, hamiltonian, which either completes the cycle if the top layer H_h' has been reached or calls the procedure continue1 (if the bypass element u was a bump cutter itself) or the procedure continue2 (if the bypass element u came from a layer H_j').

The procedure continue1 ($\mathrm{HC}_1(c_1, u_j)$, H_t') tries to grow the cycle beyond the bypass element $u = u_j$. This is done by either directly finding a new bypass element u for u_j or by recursively calling the procedure hamiltonian to find a Hamiltonian cycle on the subgraph induced by $\{c_1, u\} \cup B$, where B is the set of elements not covered by HC_1. In the first case, if the cycle can be directly grown beyond u, this is done and control is returned to the calling program (hamiltonian). If the cycle cannot be grown directly, then hamiltonian is called on the subgraph induced by $\{c_1, u_j\} \cup B$ and it will either find a critical element in this subgraph (which will be a critical element in G) or it will return a Hamiltonian cycle on this subgraph, which also contains the edge (c_1, u_j) and, so, can be combined with our original $\mathrm{HC}_1(c_1, u_j)$ into a Hamiltonian cycle of G.

The procedure `continue2` ($\mathrm{HC}_1(c_1, u)$, H'_j) tries to grow the cycle beyond the bypass element u and its layer H'_j. It tries to grow the cycle directly, as above, or by calling `bypass1` or `bypass2` or it recursively calls the procedure `hamiltonian` on the appropriately defined subgraph. The procedure `hamiltonian` then either returns some critical element in the subgraph, which is also critical in P_G, or a Hamiltonian cycle of the subgraph, which can be combined with the original $\mathrm{HC}_1(c_1, u)$ into a Hamiltonian cycle of G.

ALGORITHM 5.3.

```
procedure hamiltonian_cycle_construct ($G$)
   Input :  A cocomparability graph $G$.
   Output:  A Hamiltonian cycle in $G$ or ''$b(P_G) > 0$'' or some
                critical elements which show that $G$ has no
                Hamiltonian cycle.

   (Steps 1–3 are preprocessing steps.)

   Step 1.  Find a transitive orientation of $G^C$ and call it $P_G$.
   Step 2.  Call bumpno($P_G$).
            If $b(P_G) > 0$ or ($b(P_G) = 0$ but the dual order is not
                collapsed into a single antichain then output
                ''$G$ has no Hamiltonian cycle'' output
                ''$b(P_G) > 0$'' or any bump cutter as critical
                and STOP.
   Step 3.  ($b(P_G) = 0$ and the dual order was collapsed into a single antichain.
                Suppose that procedure bumpno has created $m + 1$ closed
                sets $A_1, A_2, \dots, A_{m+1}$ and $m$ bump cutters $u_1, u_2, \dots, u_m$ during
                its execution. Recapture all the closed sets created by bumpno,
                before collapsing.)
            For i = 0 to $m$ do;
                Put $A_{m+1-i} := A_{m+1-i} - \bigcup_{j=1}^{m-i}(A_j \cup U_j)$.
   Step 4.  Call procedure hamiltonian ($V$).
```

PROCEDURE 5.4.

```
procedure hamiltonian ($V$; $v, w$)
   Input :  A partial order $P_G$ (of a cocomparability graph $G$)
                on the set $V$. The decomposition of $P_G$ into
                closed sets $A_1$, $A_2, \dots, A_{m+1}$ and bump cutters
                $U = \{u_1, u_2, \dots, u_m\}$, a bump-optimal linear
                extension $L$ of $P_G$, the modified layers
                $H'_0, H'_1, \dots, H'_h$. A set $R$ of pairs of reserved
                independent transitions between every
                pair of consecutive layers, up to the first
                blocker layer, in the lowest closed set $A_1$.
                The blocker layers with their potentially
                critical elements identified. Optional input
                is $v, w \in H'_0$. ( When they are included,
                this signals that we want a Hamiltonian
                cycle containing edge $(v, w)$.)
```

```
Output:  A Hamiltonian cycle in $G$ or some critical
             elements in $P_G$, precluding the existence
             of such cycle.
Step 1.  If $A_1 = H'_0$ then go to Step 6.
         If $A_1$ contains no blockers then let $i$ be the
             index of its top layer $H'_i$, otherwise let i be
             the index of its lowest level blocker $H'_i$.
         Call twopaths ($H'_0 \cup H'_1 \cup \cdots \cup H'_i; R$)
         Connect the endpoints of the paths $\pi_1$ and $\pi_2$ in
             $H'_0$ and $H'_i$ to form $\mathrm{HC}_1$. Insert all the
             points missed, by both $\pi_1$ and $\pi_2$,
             in a horizontal edge in their layer
             ($\mathrm{HC}_1$ must contain a horizontal edge in every layer.)
         If the optional $v, w \in H'_0$ were specified then
             modify $\mathrm{HC}_1$ so that $(v, w)$ is an edge in $HC_1$.

There are 4 cases treated in Steps 2–5.

Step 2.  ($H'_i$ is a Type 1 blocker with potentially critical elements $x$ and $y$)
         Call bypass1 ($H'_i; x, y, \mathrm{HC}_1(c_1, x)$)
         Go to Step 7.
Step 3.  ($H'_i$ is a Type 2 blocker with potentially critical element $x$)
         Call bypass2 ($H'_i;\ x, \mathrm{HC}_1(c_1, x)$)
         Go to Step 7.
Step 4.  ($H'_i$ is the top layer of a closed set and $P_G$ is not strictly grown)
         Let $x$ be the last element from $H'_i$ in $L$ and $u_j$ its
         cover in $L$.
         Call bypass3 ($H'_i; x, u_j, \mathrm{HC}_1(c_1, x)$)
         Go to Step 7.
Step 5.  ($P_G$ is strictly grown)
         Go to Step 9.
Step 6.  ($A_1 = H'_0$) Let $x$ be the last element from $H'_0$ in $L$.
         If $|H'_0| > 1$ and $c_1$ is the point covered by $x$ in $L$
             then find a Hamiltonian cycle $\mathrm{HC}_1(c_1, x)$ in the
             (complete) graph induced by $H'_0$.
             Call bypass3 ($H'_0; x, u_1, \mathrm{HC}_1(c_1, x)$)
         else ($x$ is the only point in $H'_0$) put $\mathrm{HC}_1 := \emptyset$
             Call bypass3 ($H'_0; x, u_1, \mathrm{HC}_1$).
Step 7.  (The point $u$ is the last (bypass) element reached by our cycle)
         If $u \in H'_h$ then go to Step 8.
         If $u = u_j \in U$ then call continue1 ($\mathrm{HC}_1(c_1, u)$, $H'_t$)
             ($H'_t$ is the top layer of $A_j$)
         else ($u \in H'_j$ for some $j < h$)
             call continue2 ($\mathrm{HC}_1(c_1, u)$, $H'_j$)
         Repeat Step 7.
Step 8.  (We have reached the top layer $H'_h$)
         If $u$ is the last point in $L$ then go to Step 9.
         (At this stage $u$ cannot be the only entry point to $H'_h$, as this would
                mean $v < w$ for every $v \in H'_{h-1}$ and $w \in H'_h - u$, contradicting
                that the dual order of $P_G$ was collapsed into a single antichain
                in our preprocessing. Since we always try to make the bypass
                element $u$ the second one from its layer in $L$, this must be
                the case here too.)
            Insert the remaining elements from $H'_h - \{c_1, u\}$ in
                $\mathrm{HC}_1(c_1, u)$ between $c_1$ and $u$.
Step 9.     If $\mathrm{HC}_1$ contains all layers then output $\mathrm{HC}_1$ as a
                Hamiltonian cycle in $G$ and STOP, otherwise
                return $\mathrm{HC}_1$ to the calling procedure.
END;
```

PROCEDURE 5.5.

```
procedure bypass1 ($H'_i$; $x, y, \mathrm{HC}_1(c_1, x)$)
($H'_i$ is a Type 1 blocker with critical elements $x$ and $y$, $(c_1, x)$ is a horizontal edge
    in $\mathrm{HC}_1$ )
  Step 1.   Let $B = x|L$. Find a highest level $u \in B$ with $(c_1, u)$
                a transition. If no such $u$ exists then go to
                Step 2, otherwise go to Step 3.
  Step 2.   Find a highest level $u \in B$ with $(a, u)$ a transition
                for some $a \in H'_i - \{x, y\}$. If no such $u$ exists then
                output''$x$ and $y$ are critical'' and STOP.
            Rearrange $L$ and $\mathrm{HC}_1(c_1, x)$ so that $(a, x)$ becomes an
                edge in both (i.e., move $a$ between $c_1$ and $x$ by a
                rearrangement of Type 2).
                Put $c_1 := a$.
  Step 3.   If $u \notin U$ (i.e., $u \in H'_j$ for some $j \geq i + 1$) then modify
                $L$ (if possible) so that $u$ is the second point
                from $H'_j$ in $L$.
                (If this rearrangement of Type 1 is not possible then
                $H'_j$ must be a blocker itself and it will be determined
                later — in the procedure continue2 — whether this
                blocker can be bypassed too. This comment also
                applies in the remaining procedures where this type
                of rearrangement is attempted unsucessfully.)
  Step 4.   Let $a$ be the element covered by $u$ in $L$.
            Put $\mathrm{HC}_2(c_1, x) := \{(c_1, x), L[x, u], (u, c_1)\}$;
                $\mathrm{HC}_1(a, u) := \mathrm{HC}_1(c_1, x) \cup \mathrm{HC}_2(c_1, x) - \{(c_1, x)\}$; $c_1 := a$.
  END;
```

PROCEDURE 5.6.

```
procedure bypass2 ($H'_i$; $x, \mathrm{HC}_1(c_1, x)$)
($H'_i$ is a Type 2 blocker with potentially critical (exit) element $x$.)
  Step 1.   Let $B = x|L$. Find a highest level $u \in B$ with $(c_1, u)$
                a transition. If no such $u$ exists go to
                Step 2, otherwise go to Step 8.
  Step 2.   Find a highest level $u \in B$ with $(a, u)$ a transition
                for some $a \in H'_i - \{x, c_1, y\}$, where $y$ is the entry
                point to $H'_i$ in $L$. If no such $u$ exists go to
                Step 3, otherwise go to Step 7.
Step 3.  Find a highest level $u \in B$ with $(y, u)$ a transition.
             If no such $u$ exists then go to Step 4,
             otherwise go to Step 6.
Step 4.  Find a highest level $u \in B$ with ($(e, u)$ a transition
             to $u$, $e \in H'_{i-1}$, $e < x$ in $P_G$ and $e$ is incomparable
             to every other $v \in H'_i$). If no such $u$ exists
             then output ''$x$ is critical'' and STOP.
Step 5.  Rearrange $L$ and $\mathrm{HC}_1(c_1, x)$ by moving $e$ immediately
             before $x$ and performing the implied changes
             both in $L$ and $\mathrm{HC}_1$ in the sense of a
             rearrangement of Type 4. This results in
             a $\mathrm{HP}_1(e, x)$.
         If $u \notin U$ (i.e., $u \in H'_j$ for some $j > i$) then make
             $u$ the second point in $L$ from $H'_j$, if possible.
             Let $a$ be the element covered by $u$ in $L$ and
             $\mathrm{HP}_2(x, e) := L[x, u] \cup \{(u, e)\}$.
         Put $\mathrm{HC}_1(a, u) := \mathrm{HP}_1(e, x) \cup \mathrm{HP}_2(x, e), c_1 := a$ and go to END.
Step 6.  Find a $d \in H'_i - \{x, y\}$ with $(c, d)$ a transition for some
             $c \in H'_{i-1}$. If no such $d$ exists then output
             ''$x$ is critical'' and STOP,
             (we do not need to consider Step 4 here because the
             nonexistence of $d$ implies the nonexistence of $e$ in Step 4),
             otherwise rearrange $L$ and $\mathrm{HC}_1$ by moving $y$
             immediately before $x$ in $L$, making $d$ the
             entry point to $H'_i$ in $L$ and all the implied
             changes like in a rearrangement of
             Type 2.
         Put $c_1 := y$ and go to Step 8.
Step 7.  Move $a$ between $c_1$ and $x$ in both $L$ and $\mathrm{HC}_1$ and put
             $c_1 := a$.
Step 8.  If $u \notin U$ (i.e., $u \in H'_j$ for some $j > i$) then make $u$
             the second element from $H'_j$ in $L$, if possible.
         Let a be the element covered by $u$ in $L$.
             Let $\mathrm{HC}_2(c_1, x) := \{(c_1, x), L[x, u], (u, c_1)\}$.
             Put $\mathrm{HC}_1(a, u) := HC_1(c_1, x) \cup \mathrm{HC}_2(c_1, x) - \{(c_1, x)\}; c_1 := a$.
END;
```

PROCEDURE 5.7.

```
procedure bypass3 ($H'_i$; $x, u_k, \mathrm{HC}_1(c_1, x)$)
($H'_i$ is the top layer in the closed set $A_k$, $x$ is its last element in $L$, $x$ covers $c_1$ in $L$ and
    $u_k$ is the bump cutter between $A_k$ and $A_{k+1}$.)
  Step 1.  If $A_1 = H'_i = \{x\}$ then find a highest level
               $u \in V - \{x, u_1\}$ with $(x, u)$ a transition
               ($u$ must exist since the dual order was collapsed),
               otherwise go to Step 4.
               If $u \in \{u_2, u_3, \ldots, u_m\}$ go to Step 3,
               otherwise go to Step 2.
  Step 2.  ($u \in H'_j$ for some $j > i$): Modify $L$ (if possible)
               so that $u$ is the second point from $H'_j$ in $L$
               (rearrangement of Type 1).
```

```
Step 3.  Let $a$ be the element immediately preceding $u$ in $L$.
             Put $\mathrm{HC}_1(a,u) := \{L[x,u], (u,x)\}$, $c_1 := a$ and go to END.
Step 4.  (HC$_1$ has reached $H_i'$ but not beyond.)
         Find a highest level
             $u \in A_{k+1} \cup u_{k+1} \cup A_{k+2} \cup u_{k+2} \cup \cdots \cup A_{m+1}$
             with $(c_1,u)$ a transition from $H_i'$. If no such $u$
             exists then go to Step 5. If $u \in H_j'$ for some $j$
             then modify $L$ (if possible) to make $u$ the
             second point from $H_j'$ in $L$ (rearrangement of Type 1).
         Put $\mathrm{HC}_1(a,u) := \mathrm{HC}_1(c_1,x) \cup L[x,u] \cup \{(u,c_1)\} - \{(c_1,x)\}$
             and $c_1 := a$, where $a$ is the point immediately
             before $u$ in $L$. Go to END.
Step 5.  Find a highest level $u \in A_{k+1} \cup u_{k+1} \cup \cdots \cup A_{m+1}$ with
             $(c,u)$ a transition for some $c \in H_i' - \{x,y\}$, where
             $y$ is the entry point to $H_i'$ in $L$. If no such
             $u$ exists then go to Step 6. If $u \in H_j'$ for some
             $j$ then modify $L$ (if possible) to make $u$ the
             second point from $H_j'$ in $L$ (rearrangement of Type 1).
         Move $c$ between $c_1$ and $x$ both in $L$ and $\mathrm{HC}_1(c_1,x)$ .
         Put $\mathrm{HC}_1(a,u) := \mathrm{HC}_1(c,x) \cup L[x,u] \cup \{(u,c)\} - \{(c,x)\}$
             and $c_1 := a$, where $a$ is the point immediately
             preceding $u$ in $L$. Go to END.
Step 6.  Find a $u \in A_{k+1} \cup u_{k+1} \cup \cdots \cup A_{m+1}$ with $(y,u)$ a transition.
             If no such $u$ exists then go to Step 7.
             If $|H_i'| > 2$ then modify $L$ and $\mathrm{HC}_1(c_1,x)$, so that
             a $d \in H_i' - \{x,y\}$ is used as entry point to $H_i'$
             in $L$ and $\mathrm{HC}_1$ and move $y$ immediately before
             $x$ in both $L$ and $\mathrm{HC}_1$ (like in a rearrangement of Type 2),
             otherwise if no such $d$ exists then output
             ''$x$ and $y$ are critical'' (by Lemma 3.8(1) )
             and STOP.
         Put $c_1 := y$. If $u \in H_j'$ for some j then modify $L$
             (if possible) to make $u$ the second point from
             $H_j'$ in $L$ (rearrangement of Type 1).
         Put $\mathrm{HC}_1(a,u) := \mathrm{HC}_1(c_1,x) \cup L[x,u] \cup \{(u,c_1)\} - \{(c_1,x)\}$
             and $c_1 := a$, where $a$ is the point immediately
             preceding $u$ in $L$. Go to END.
Step 7.  ($x$ is the only exit point from $H_i'$):
         Find a highest level $u \in A_{k+1} \cup\ u_{k+1} \cup \cdots \cup A_{m+1}$ with
             $(e,u)$ a transition for some $e \in H_{i-1}'$ with the
             properties $e < x$ and $e$ incomparable to every
             point between $e$ and $x$ in $L$. If no such $e$
             and $u$ exist then output ''$x$ is critical''
             (by Lemma 3.7) and STOP. Otherwise modify $L$
             so that $e$ moves up to become the immediate
             predecessor of $x$ in $L$ and correspondingly,
             $\mathrm{HC}_1(c,x)$ is replaced by $\mathrm{HP}_1(e,x)$ (rearrangement of
             Type 4). If $u \in H_j'$ for some $j$ then modify $L$
```

```
            (if possible) to make $u$ the second point from
            $H'_j$ in $L$ (rearrangement of Type 1).
        Put $\mathrm{HC}_1(a,u) := \mathrm{HP}_1(e,x) \cup L[x,u] \cup \{(u,e)\}$ and $c_1 := a$,
            where $a$ is the point covered by $u$ in $L$.
END;
```

PROCEDURE 5.8.

```
procedure continue1 ($\mathrm{HC}_1(c_1,u_j)$, $H'_t$)
(Our cycle has reached the bump cutter $u_j$, using it as a bypass element. $H'_t$ is
the top layer of $A_j$, $c_1 \in H'_t$ is the point covered by $u_j$ in $L$. )
    Step 1.  Let $B = u_j|L$. Find a highest level $u \in B$ with $(c_1,u)$
                 a transition. If no such $u$ exists then go to
                 Step 3. If $u \notin U$ then rearrange $L$ so that $u$ is
                 the second point from its layer in $L$
                 (if possible).
    Step 2.  Put $\mathrm{HC}_2(c_1,u_j) := L[c_1,u] \cup \{(u,c_1)\}$, $\mathrm{HC}_1(a,u) := \mathrm{HC}_1(c_1,u_j) \cup$
                 $\mathrm{HC}_2(c_1,u_j) - \{(c_1,u_j)\}$, $c_1 := a$, where $a <_L u$.
                 Go to END.
    Step 3.  If $H'_t = \{c_1\}$ then output ''$u_j$ is critical'' (there is
                 no transition from $H'_t$ up if we delete $u_j$) and STOP.
             Find a highest level $u \in B$ and $c \in H'_t - \{c_1\}$ with $(c,u)$
                 a transition. If no such $u$ exists then output
                 ''$u_j$ is critical'' and STOP.
             If $u \notin U$ then rearrange $L$ so that $u$ is the second
                 point from its layer in $L$ (if possible).
                 If $c$ is not the entry point to $H'_t$ in $L$ then
                 go to Step 4.
             Rearrange $L$ and $\mathrm{HC}_1(c_1,u_j)$ to make $c$ the exit
                 point from $H'_t$ in $L$ in the sense of Lemma 4.6.
                 (The resulting $\mathrm{HC}_1$ will have $(c,u_j)$ as edge.)
             Go to Step 5.
    Step 4.  Rearrange $L$ and $\mathrm{HC}_1(c_1,u_j)$ to make $c$ the exit point
                 from $H'_t$ in $L$ and $(c,u_j) \in \mathrm{HC}_1$.
    Step 5.  Put $\mathrm{HC}_2(c,u_j) := \mathrm{HC}_1$.
             Call hamiltonian ($\{c,u_j\} \cup B$) (the returned $\mathrm{HC}_1$ will
             contain $(c,u_j)$).
                 Put $\mathrm{HC} := \mathrm{HC}_1 \cup \mathrm{HC}_2(c,u_j) - \{(c,u_j)\}$.
             Output HC as a Hamiltonian cycle for $G$ and STOP.
    END;
```

PROCEDURE 5.9.

```
procedure continue2 ($\mathrm{HC}_1(c_1,u)$, $H'_j$)
(Our cycle $\mathrm{HC}_1(c_1,u) = \mathrm{HC}_u$ has reached $u \in H'_j$ as a bypass element, $|H'_j| \geq 2$
    must hold.)
    Step 1.  Let $B = u|L$. Find a highest level $w \in B$ with $(c_1,w)$
                 a transition.
             If no such $w$ exists then go to Step 2. If $w \notin U$
                 try to make it the second point in $L$ from
                 its layer, (if possible). Let
```

```
                $\mathrm{HC}_2(c_1, u) := L[c_1, w] \cup \{(w, c)\}$,
                $\mathrm{HC}_1(a, w) := \mathrm{HC}_1(c_1, u) \cup \mathrm{HC}_2(c_1, u) - \{(c_1, u)\}, c_1 := a$,
                where $a$ is the element covered by $w$ in $L$.
                Go to END.
Step 2.     If $u$ is the second point in $L$ from $H'_j$ then go to
            Step 3. ($u$ could not be made the second point from $H'_j$ in $L$,
                so $u$ must be the entry or exit for $H'_j$ and $H'_j$ must be a blocker.)
            If $u$ is the potentially critical exit element from
            the Type 2 blocker $H'_j$ then
                call bypass2 ($H'_j$; $u, \mathrm{HC}_1(c_1, u)$).
                Go to END.
            If $H'_j$ is a Type 1 blocker and $u$ is incomparable to
                every $v \in H'_{j-1}$ in $P_G$ then rearrange $L$ and $\mathrm{HC}_1$
                so that $u$ becomes the exit point from $H'_{j-1}$
                in $L$.
                Call bypass1($H'_j$; $u, y, \mathrm{HC}_1(c_1, u)$), where $y$ is the
                    other possible exit from $H'_j$.
                Go to END.
            ($u$ is the entry point to $H'_j$ in $L$, $|H'_j| > 2$ by Lemma 4.7 and $u$
                could not be made second from $H'_j$ in $L$.)
                Output ``$u$ is critical'' (by Lemma 3.6) and STOP.
Step 3.     If $H'_j$ is a Type 2 blocker and $|H'_j| > 2$ then go to
                Step 4.
            Call hamiltonian ($H'_j \cup B; c_1, u$) (It returns an $\mathrm{HC}_1(c_1, u)$.)
            Put $\mathrm{HC} := \mathrm{HC}_u \cup \mathrm{HC}_1(c_1, u) - \{(c_1, u)\}$, output HC for $G$
                and STOP.
Step 4.     ($H'_j$ is a Type 2 blocker and $u$ is neither the entry point to it
                nor the exit from it in $L$.)
            Let $x'$ be the (only) exit from $H'_j$ in $L$. Find a
                highest level $w \in B$ with $(a, w)$ a transition for
                some $a \in H'_j - \{u, x'\}$. If no such $w$ exists then
                go to Step 5. Rearrange $L$ and $\mathrm{HC}_u$ so that $u$
                covers $a$ in $L$ and $(a, u) \in \mathrm{HC}_u$ (Type 2 rearrangement).
            Call hamiltonian ($\{a, u\} \cup B; a, u$) ($\mathrm{HC}_1(a, u)$ is the cycle
                returned.)
            Put $\mathrm{HC} := \mathrm{HC}_u \cup \mathrm{HC}_1(a, u) - \{(a, u)\}$, output HC for
                $G$ and STOP.
Step 5.     Find a highest level $w \in B$ with $(e, w)$ a transition
                between the $j$th and (j+1)st layer in $P_G - x'$
                (i.e., $e < x'$, $e \in H'_{j-1}$, and $e$ incomparable to every point in
                $H'_j - \{x'\}$). If no such $w$ and $e$ exist then output
                ``$x'$ is critical'' and STOP.
            Rearrange $L$ and $\mathrm{HC}_u$ so that $e$ is covered by $u$
                in $L$ (make $e$ the exit from $H'_{j-1}$ and $u$ the
                entry to $H'_j$ in $L$), and $(e, u)$ becomes an edge
                in $\mathrm{HC}_u$ (Type 3 rearrangement).
                Put $B := u|L$, using the rearranged $L$.
```

```
        Call hamiltonian ($\{e,u\} \cup B; e, u$) ($\mathrm{HC}_1(e,u)$ is returned.)
        Put $\mathrm{HC} := \mathrm{HC}_u(e,u) \cup \mathrm{HC}_1(e,u) - \{(e,u)\}$, output HC for $G$
            and STOP.
END;
```

Let us consider the application of Algorithm 5.3 to the example shown in Fig. 7. Although the bumpno algorithm creates, during its execution, the closed sets $A_1, A_2, \ldots, A_6$ and bump cutters $u_1, u_2, \ldots, u_5$, eventually all these get collapsed into a single antichain. (The transition (4,13) collapses $A_1 \cup A_2 \cup u_1 \cup u_2 \cup H_3' \cup H_4'$ into a single antichain A_3, which is grown to include H_5'. Following this, the transition (26, 31) collapses $A_6 \cup u_5 \cup A_5$ into a single antichain A_5, which has the endpoint of the transition (14, 30) between A_3 and A_6, causing the whole dual order to collapse into a single antichain $A_1 = \{1, 2, \ldots, 33\}$.) In Step 3 Algorithm 5.3 recaptures the original $A_1, A_2, \ldots, A_6$, which are shown in Fig. 7. Suppose the optimal linear extension created by the algorithm bumpno is $L = 1, 2, 3, 5, 4, 8, 7, 6, 9, 10, 11, 13, 12, 14, 15, 16, 19, 18, 17, 20, 21, 22, 24, 23, 27, 26, 25, 30, 29, 28, 31, 32, 33$. The algorithm bumpno would also identify the Type 1 blocker H_7' (with potentially critical elements 20 and 22) and the Type 2 blockers H_2' and H_4' with potentially critical elements 6 and 12, respectively.

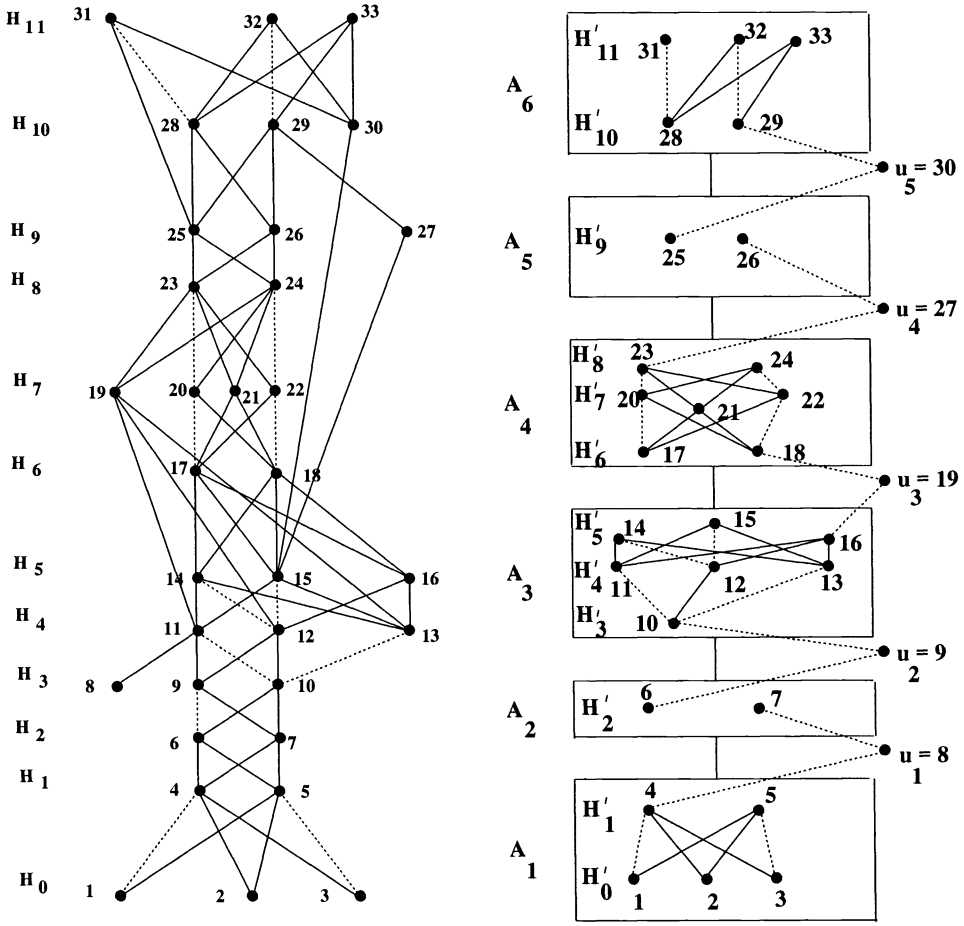


FIG. 7. *An example for the application of Algorithm* 2.

Applying `hamiltonian` to P_G starts by calling `twopaths` ($H'_0 \cup H'_1$; R), where the reserved transitions are $R = \{(1,4),(3,5)\}$. This returns HC$_1$ = (1, 2, 3, 5, 4, 1). Since H'_1 is not a blocker, it calls `bypass3` (H'_1; 4, 8, HC$_1$(5, 4)) to bypass the bump cutter $u_1 = 8$. This finds $u = 13$ as the highest level bypass element with the transition (5, 13) and enlarges HC$_1$ into HC$_1$ (11, 13) = (1, 2, 3, 5, 13, 11, 10, 9, 6, 7, 8, 4, 1). After returning to `hamiltonian`, since the bypass element u is not a bump cutter, it calls `continue2` (HC$_1$ `(11, 13),` H'_4). This identifies $B = 13 \mid L = \{12, 14, 15, \ldots, 33\}$. No transition exists from $c_1 = 11$ to B, H'_4 is a Type 2 blocker with potentially critical $x' = 12$, and no transition exists from $H'_4 - \{11, 12\} = \{13\}$ to B. The element $e = 10$ is identified as the start of the transition (10, 14) to $w = 14 \in B$. By a Type 3 rearrangement of L we make $e = 10$ the exit from H'_3 and $u = 13$ the entry to H'_4. This results in L = 1, 2, 3, 5, 4, 8, 7, 6, 9, 10, 13, 11, 12, …, with no change after 12, and HC$_1$ = (1, 2, 3, 5, 13, 10, 9, 6, 7, 8, 4, 1). The element 11 gets added to B. After this `hamiltonian` ($\{10, 13\} \cup B$; 10, 13) is called, with L_1 = 10, 13, 11, 12, …, continued as L after 12. This calls `twopaths` ($H'_3 \cup H'_4$; R), where the reserved transitions are $R = \{(10,11),(10,13)\}$. This returns HC$_2$ = (10, 11, 12, 13, 10). After this `bypass2` (H'_4; 12, HC$_2$(11, 12)) is called, which identifies the transition (10,14) to bypass x = 12, with u = 14 the bypass element and $e = 10$ moving up to replace x in $P_G - x$. By a rearrangement of Type 4, $e = 10$ is moved immediately before $x = 12$ in L_1 and HC$_2$, "taking" also 13 with it (to keep the edge (10, 13)), resulting in L_1 = 11, 13, 10, 12, … and HP_2(10, 12) = (10, 13, 11, 12). (Note that (10, 12) is a bump in L_1, and correspondingly (10, 12) is not an edge in G.) The bypass element $u = 14$ is not the second point from H'_5 in L_1, so L_1 is rearranged (by a Type 1 rearrangement) into L_1 = 11, 13, 10, 12, 15, 14, 16, 19, …. HP$_3$(12, 10) $= L_1[12, 14] \cup (14, 10) =$ (12, 15, 14, 10) is constructed and is combined into HC$_2$(15, 14) $= HP_2$(10, 12) ∪ HP$_3$(12, 10) = (10, 13, 11, 12, 15, 14, 10). Returning to `hamiltonian, continue2` (HC$_2$(15, 14), H'_5) is called. B is set to $B = \{16, 19, 17, 18, 20, 21, \ldots, 33\}$. The bypass element $u = 14$ is second from H'_5 in L_1, so `hamiltonian` ($H'_5 \cup B$; 14, 15) is called with L_2 = 15, 14, 16, 19, …. This constructs HC$_3$ = (14, 15, 16, 14) and calls `bypass3` (H'_5; 16, 19, HC$_3$(14, 15)). This finds the highest level bypass element $u = u_5 = 30$ with the transition (14, 30). Using this u, we enlarge our cycle into HC$_3$(14, 15) = (14, 15, 16, 19, 18, 17, 20, 21, 22, 24, 23, 27, 26, 25, 30, 14) and return to `hamiltonian`, which calls `continue1` (HC_3(25, 30), H'_9). (Note that 27 was also available to bypass $u_3 = 19$, but 30 being from a higher layer, choosing 30 speeds up the algorithm. We also bypassed the Type 1 blocker H_7 in the process). B is set to $B = \{29, 28, 31, 32, 33\}$. The bypass element $u = 31$ with the transition (26, 31) is identified. L_2 is rearranged into L_2 = 15, 14, 16, 19, 18, 17, 20, 21, 22, 24, 23, 27, 26, 25, 30, 28, 29, 32, 31, 33 to make u = 31 the second point from H'_{11} in L_2 . As $c = 26$ is the entry point to H'_9 in L_2, we rearrange L_2 , in the sense of Lemma 4.6, into L_2 = 15, 14, 16, 19, 18, 17, 20, 21, 22, 24, 23, 27, 25, 26, 30, 28, 29, 32, 31, 33 to make 26 the exit point from H'_9. This rearrangement results in HC$_3$(26, 30) = (14, 15, 16, 19, 18, 17, 20, 21, 22, 24, 23, 27, 25, 26, 30, 14). `hamiltonian` ($\{26, 30\} \cup B$) is called with L_3 = 26, 30, 28, 29, 32, 31, 33, which, after a call to `bypass3` ($\{26\}$; 26, 30, ϕ), returns HC$_4$(26, 30) = (26, 30, 28, 29, 32, 33, 31, 26). All is left is to combine the cycles constructed:

$$\begin{aligned}
\mathrm{HC}_5 &= \mathrm{HC}_3(26,30) \cup \mathrm{HC}_4(26,30) - \{(26,30)\} \\
&= (14, 15, 16, 19, 18, 17, 20, 21, 22, 24, 23, \\
&\qquad 27, 25, 26, 31, 33, 32, 29, 28, 30, 14) \\
\mathrm{HC}_6 &= \mathrm{HC}_2(14,15) \cup \mathrm{HC}_5 - \{(14,15)\} \\
&= (10, 13, 11, 12, 15, 16, 19, 18, 17, 20, \\
&\qquad 21, 22, 24, 23, 27, 25, 26, 31, 33, \\
&\qquad 32, 29, 28, 30, 14, 10)
\end{aligned}$$

$$\begin{aligned}\mathrm{HC} &= \mathrm{HC}_1(10,13) \cup \mathrm{HC}_6 - \{(10,13)\}\\ &= (1,2,3,5,13,11,12,15,16,19,18,17,\\ &\qquad 20,21,22,24,23,27,25,26,31,33,\\ &\qquad 32,29,28,30,14,10,9,6,7,8,4,1)\end{aligned}$$

THEOREM 5.10. *Algorithm* 5.3 *solves the* Hamiltonian Cycle Construction problem *for a cocomparability graph G in $O(hn^2)$ time and $O(n^2)$ space, where n is the number of vertices in G and h is the size of a largest independent set in G.*

Proof. The correctness of the algorithm follows from Lemmas 3.4 to 3.9, 4.5 to 4.7 and Theorem 4.8. The only change in the algorithm, from the development of the previous results, is that instead of using the bumpno algorithm on $P_G - x$ to find a bypass element u for the potentially critical element x, we look for a *highest level* bypass element u *directly*. This means that the bypass element we find may not be the same as what bumpno would have found. The reason for looking for a highest level bypass element is that this way our Hamiltonian cycle is grown faster, by possibly reaching higher levels in fewer iterations. Once we have grown the Hamiltonian cycle into an HC_u covering $G|u$, for any bypass element u, by Lemma 3.4, it follows that the subposet $L|u$ cannot have critical elements, even though we may not have directly tested all its potentially critical elements.

For the space complexity of the algorithm we note that all the information needed and generated by bumpno can be clearly stored in $O(n^2)$ space.

To prove the claimed time complexity we make the following observations: G^{C} and the transitive orientation P_G can be found in $O(n^2)$ time [18]. The bumpno algorithm for P_G needs at most $O(n^2)$ time [10]. It is clear that P_G has at most $O(h)$ blockers or bump cutters, so the procedures bypass1, bypass2, bypass3, continue1 *and* continue2 can be called at most $O(h)$ times. A single execution of any of these can clearly be done in $O(n^2)$ time. □

We note that a careful implementation of the algorithm could result in a version with $O(n^2)$ time complexity, by exploiting the fact that each transition present in P_G would have to be looked at at most two times (once when searching for an appropriate transition, and possibly the second time when rearrangements are performed). We omitted this from the statement of the theorem, because proving it would require a much more detailed statement of the algorithm. Finally, we also note that the Hamiltonian cycle constructed by the algorithm is "layered" in a certain sense: it enters each layer H_i' at most twice.

6. Appendix.

ALGORITHM 6.1.

```
algorithm bumpno($P$)
    Step 1:  (Initialization):
                 Let $H_0, \dots, H_h$ be the initial layers of $P$. Put
                 $H_i' := H_i (i = 0, \dots, h), l := 1, k := 0$. open($A_l$), put $k := 1$.
    Step 2:  (Termination condition):
                 If $k = h$, close($A_l, a$) with any maximal $a \in A_l$ and
                 terminate. Then the concatenation $L = L_1 U_1 L_2 \dots$
                 $U_{l-1} L_l$ is an optimal linear extension and
                 $A_1, \dots, A_l$ and $U$ define the contained
                 generalized weak order $P^*$.
    Step 3:  (Search for transitions from $H_{k-1}'$):
                 Search for transitions $(a_{k-1}, u_{k-1})$ from $H_{k-1}'$ to
                 $H_j'$ with $j \geq k$ according to RULE 1, RULE 2, and
                 RULE 3. There are 4 cases that are treated in
                 Steps 4-7.
```

```
Step 4:  (There are two or more transitions into $H'_k$ with
           different endpoints): Then grow($A_l$). If there
           is a transition $(a,b)$ from some $a \in \bigcup_{i=1}^{l-1} A_i$ to
           some $b \in H'_k$ (which can only happen if $l > 1$),
           then collapse($A_l$) and put $l := t$ (a value returned
           by collapse). Put $k := k+1$ and go to Step 2.
Step 5:  (All transitions into $H'_k$ have the same endpoint):
           Let $(a_{k-1}, u_{k-1})$ be such a transition. Then
           close($A_l, a_{k-1}$), put $H'_k := H'_k - \ \{u_{k-1}\}, U_l := \{u_{k-1}\}, l := l+1$,
           and open($A_l$). Then open returns $A_l = H'_k$. If
           there is a transition $(a,b)$ from some $a \in \bigcup_{i=1}^{l-1} A_i$
           to some $b \in H'_k$ then collapse($A_l$) and put $l := t$
           (a value returned by collapse). Put $k := k+1$
           and go to Step 2.
Step 6:  (There are only transitions into higher layers $H'_j$, $j > k$) :
           Choose such a transition $(a_{k-1}, u_{k-1})$ with
           $u_{k-1} \in H_j$ according to RULE 1 and RULE 2.
           Then close($A_l, a_{k-1}$), put $H'_j := H'_j - \{u_{k-1}\}$,
           $U_l := \{u_{k-1}\}$, open($A_{l+1}$), put $l := l+1, k := k+1$
           and go to Step 2.
Step 7:  (There is no transition at all): Then close($A_l, a$)
           with any maximal $a \in A_l$ , put $U_l := \phi$ (this means
           that a bump is created), open($A_{l+1}$) put $l : l+1$,
           $k := k+1$, and go to Step 2.
```

The change in bumpno, in comparison to the original algorithm in [10], is the addition of RULE 3 for breaking ties between short transitions. Clearly this does not affect its correctness or complexity. The procedure open initializes the new antichain and creates data structures necessary for storing the information.

PROCEDURE 6.2.

```
procedure open($A_l$).
        Put $A_l^1 := \phi, A_l^2 := H'_k$ (where $U_0 = \phi$), and $A_l := A_l^1 \cup A_l^2$ .
        Create a list list($A_l^2$) for transitions to be stored,
        create a list list($A_l^1$) for storing a linear exten-
        sion $L_l^1$ of $A_l^1$ (for the case that $A_l^1$ becomes nonempty).
```

The procedure grow(A_l) is rather simple and just keeps track of the information needed to construct the linear extension of A_l upon closing. Note that grow is the only step in the algorithm where A_l^2 is enlarged.

PROCEDURE 6.3.

```
procedure grow($A_l$).
        Put the two transitions $(a_{k-1}, u_{k-1}), (a'_{k-1}, u'_{k-1})$
        with $u_{k-1} \neq u'_{k-1}$ on list($A_l^2$). Put $A_l^2 := A_l^2 \cup H'_k$.
```

The procedure `close` constructs a linear extension of A_l according to the information stored in list(A_l^1) and list(A_l^2).

PROCEDURE 6.4.

```
procedure close($A_l, a$).
          Construct a linear extension $L_l$ of $A_l$ without
          bumps with last element $a$.
          Put the set $A_l$ on list closedsets. If $k < h$ then put
          the bump cutter $u_l$ on list bumpcutters.
          (These lists are constructed because they are needed
          to reconstruct collapsed closed sets.)
```

The procedure `collapse` is applied when the layer H_k' that is to be added to A_l or A_{l+1} is not above some previous antichain A_i. This can only occur if H_k' contains endpoints of transitions from previous layers contained in previous antichains. Loosely speaking, `collapse` joins as many antichains A_i and associated sets U_i below A_l to the A_l^1 — part of A_l as necessary to ensure that $A_1 < A_2 < \ldots < A_l$ holds at all the time and updates the associated linear extension L_l^1 of A_l^1. Note that `collapse` is the only part in the algorithm where A_l^1 is enlarged.

In looking for the *range* over which the collapsing has to take place, a certain closure operation is used. First, all transitions (a, b) to all $b \in H_k'$ are considered and if $a \in A_i$, $i < l$, then $A_i \cup U_i$ is added to A_l. Then the same is done for transitions (a, b) with $b \in U_j$ for all added U_j, iterating in this fashion through $l-1, l-2, \ldots$ until we find the first (largest) t for which there is no transition from $\bigcup_{j=1}^{t-1} A_j$ to $A_l^{\text{new}} = A_l^{\text{old}} \cup U_{l-1} \cup A_{l-1} \cup \cdots \cup U_t \cup A_t$.

PROCEDURE 6.5.

```
procedure collapse($A_l$).
          Find, in the order $l-1, l-2, \ldots$ the first index $t$
          such that if $(a,b)$ is a transition with $b \in H_k' \cup (\bigcup_{i=t}^{l-1} U_i)$
          then $a \notin \bigcup_{j=1}^{t-1} A_j$. Enlarge $A_l^1$ by putting $A_l^1 := A_l^1 \cup \{\bigcup_{i=t}^{l-1}(A_i$
          $\cup U_i)\}$ and update the linear extension $L_l^1$ by letting
          $L_t U_t L_{t+1} U_{t+1} \ldots L_{l-1} U_{l-1}$ precede the old $L_l^1$.
          Store the new $L_l^1$ in list($L_l^1$). Finally, return $t$.
```

It was shown in [10] that t can be characterized as the largest index fulfilling $\bigcup_{j=1}^{t-1} A_j <_P \bigcup_{j=t}^{l} (A_j \cup U_j)$. For more detail and examples we refer the reader to [10].

REFERENCES

[1] C. BERGE, *Graphs*, North-Holland, Amsterdam, 1985.

[2] A. A. BERTOSSI, *The edge Hamiltonian path problem is NP-complete*, Inform. Process. Lett., 13 (1981), pp. 157–159.

[3] P. DAMASCHKE, *The Hamiltonian circuit problem for circle graphs is NP-complete*, Inform. Process. Lett., 32 (1989), pp. 1–2.

[4] P. DAMASCHKE, J. S. DEOGUN, D. KRATSCH, AND G. STEINER, *Finding Hamiltonian paths in cocomparability graphs using the bump number algorithm*, Order, 8 (1992) pp. 383–391, 1992.

[5] U. Faigle and R. Schrader, *A combinatorial bijection between linear extensions of equivalent orders*, Discrete Math., 58 (1986) pp. 295–301.

[6] M. R. Garey, D. S. Johnson, and R. E. Tarjan, *The planar Hamiltonian circuit problem is NP-complete*, SIAM J. Comput., 5 (1976), pp. 704–714.

[7] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[8] D. Gouyou-Beauchamps, *The Hamiltonian circuit problem is polynomial for 4-connected planar graphs*, SIAM J. of Comput., 11 (1982), pp. 529–539.

[9] M. Habib, *Comparability invariants*, Ann. Discrete Math., 23 (1984), pp. 371–386.

[10] M. Habib, R. H. Möhring, and G. Steiner, *Computing the bump number is easy*, Order, 5 (1988), pp. 107–129.

[11] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, *Hamiltonian paths in grid graphs*, SIAM J. of Comput., 11 (1982), pp. 676–686, 1982.

[12] D. S. Johnson, *The NP-completeness column: An ongoing guide*, J. Algorithms, 6 (1985), pp. 434–451.

[13] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computer Communications, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[14] M. Keil, *Finding Hamiltonian circuits in interval graphs*, Inform. Process. Lett., 20 (1985), pp. 201–206.

[15] D. Kratsch and L. Stewart, *Domination on cocomparability graphs*, SIAM J. Discrete Math., to appear.

[16] M. S. Krishnamoorthy, *An NP-hard problem in bipartite graphs*, SIGACT News, 7 (1976), p. 26.

[17] A. A. Schäffer and B. B. Simons, *Computing the bump number with techniques from two-processor scheduling*, Order, 5 (1988), pp. 131–141.

[18] J. Spinrad, *Transitive orientation in $O(n^2)$ time*, in 15th STOC Proceedings, 1983, pp. 457–466.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 553–562, June 1994

PACKET TRANSMISSION IN A NOISY-CHANNEL RING NETWORK*

VICTOR PESTIEN†, S. RAMAKRISHNAN†, AND DILIP SARKAR†

Abstract. Assume that n stations, each with a buffer to hold only one packet at a time, are connected as a ring and that data packets are transmitted counterclockwise. A station will attempt to transmit a packet to the next station only if (i) it has a packet to send and (ii) the next station's buffer is empty. The communication channels connecting the stations are noisy, and there is a fixed probability p $(0 < p < 1)$ of error-free transmission of a packet from one station to the next in one attempt.

An exact expression for the long-run average time for a packet to go around the ring is derived. (A special case of this answers a question raised by Berman and Simon in [*Proc. 20th ACM Symp. on Theory of Computing*, ACM Press, 1988, pp. 66–77].) For fixed n and p, the throughput of the system is maximum when the number of packets is an integer closest to $n/2$.

Key words. ring network, packet communication, equilibrium distribution, Markov chain

AMS subject classifications. primary: 68M10; secondary: 60J10, 94A05

1. Introduction.

Local area networks are a subject of increasing importance for efficient data communication. A ring network is a particularly attractive local area network, since its architecture uses simple control software and interfaces. In a unidirectional ring network, the host computers are connected to the network via ring interface hardware. Each interface is called a station in the ring. A station transmits its message to the next station in the ring, and the message circulates around the network until it reaches the destination station where it is forwarded to the host computer. The interface hardware can identify messages intended for its host. A host may transmit its packet by passing it to its attached station using one of several standard protocols (see Tanenbaum [6]).

In this paper, we study the asymptotic average rate of transmission of packets of messages in a ring network. We assume that a station can store only one packet of a message at a time in its buffer. Thus, a station can transmit a packet to the next station if the next station has an empty buffer. During transmission, a packet can become corrupted in the channel. In other words, the transmission channel is not noise-free. Assume that there is a probability p of error-free transmission in one attempt. If a packet is corrupted during transmission, it must be retransmitted. Also assume that all the stations synchronously attempt to transmit packets.

We assume that there is a fixed interval of time between consecutive transmission attempts. Within this interval, each station will determine the status of the buffer of the next station, attempt transmission, and receive acknowledgment. We further assume that the transmission of the acknowledgment is error-free. This assumption is harmless since the size of the acknowledgment packet is very small compared to the size of a packet of information, and hence it can be encoded using some error-correcting code.

This paper illustrates how the theory of Markov chains could be naturally used to model and analyze a computing problem. In §2 the system is formalized as a Markov chain and its equilibrium distribution is obtained. This distribution is then used in §3 to calculate the asymptotic average cycle time of a packet. In §4, the condition for maximizing throughput is established. Throughput of the system is the expected number of packets successfully transmitted from one station to the next in one unit of time. Section 5 gives some concluding remarks and discusses connections with the work of Berman and Simon [1].

2. Stationary probability distribution.

Consider n *stations* arranged in a circle. We assume that the stations are numbered $0, 1, \dots, n-1$. Also, for any integer j, by *station* j, we

*Received by the editors July 12, 1989; accepted for publication (in revised form) January 19, 1993.

†Department of Mathematics and Computer Science, University of Miami, Coral Gables, Florida 33124-4250.

shall mean the station j mod n. Suppose there are k packets ($k < n$) that are to be transmitted counterclockwise around the circle under the following conditions:

(i) packets can move only at discrete times $t = 1, 2, \ldots$;

(ii) at every time instant t, each packet is at some station and there is at most one packet at each station;

(iii) at every time instant t and for every integer j, a packet located at station j can move only if station $(j + 1)$ does not have a packet; if it *can* move, it *will* move, independently of other packets that can move, with probability p $(0 < p < 1)$ to station $(j + 1)$, and it will stay at station j with probability $(1 - p)$.

We want to calculate the long-run average speed with which a packet goes around the circle. We shall formalize this stochastic process as a discrete-time Markov chain with stationary transition probabilities. (All the definitions and results from the theory of Markov chains that we shall need can be found in Chung [2] or Feller [3].)

Given a time instant, we say that the *local state* of a station is

$$\begin{cases} 1 & \text{if it has a packet at that instant,} \\ 0 & \text{if it does not.} \end{cases}$$

The state space S of our Markov chain consists of n-tuples $s = (s_0, \ldots, s_{n-1})$, where s_j is the local state of station j. Formally,

$$S = \{s \in \{0, 1\}^n : \text{ exactly } k \text{ coordinates of } s \text{ are } 1\}.$$

For $s \in S$ and for any integer j, by s_j we shall mean the $(j \bmod n)$th coordinate of s. For $s \in S$ and any integer j, we say that j is an *opportunity station* of state s if $s_j s_{j+1} = 10$; and we say that j is a *post-opportunity station* of state s if $s_j s_{j+1} = 01$. An opportunity station j is one that has a packet that can move in a one-step transition to station $j + 1$ since there is no packet at station $j + 1$, while a post-opportunity station is one from which a packet may have just moved to station $j + 1$. Let

$$\mathrm{OPP}(s) = \{j : 0 \le j \le n - 1 \text{ and } j \text{ is an opportunity station of state } s\}$$

and

$$\mathrm{POSTOPP}(s) = \{j : 0 \le j \le n - 1 \text{ and } j \text{ is a post-opportunity station of state } s\}.$$

The states that can result from a state s after a one-step transition are in one-to-one correspondence with subsets of OPP(s), the subset identifying the stations from which a packet actually moves. If $A \subseteq \mathrm{OPP}(s)$, the state s^A that will result from s when A is the set of stations from which packets have just moved is described formally as follows:

(i) for $j \in A$, $s_j^A s_{j+1}^A = 01$;

(ii) if s_j^A is not specified by (i), then $s_j^A = s_j$.

The entries of the transition matrix P of our Markov chain are defined by

$$P(s, s^A) = p^{|A|}(1 - p)^{|\mathrm{OPP}(s)| - |A|}, \tag{2.1}$$

for $s \in S$ and $A \subseteq \mathrm{OPP}(s)$. ($|\cdot|$ denotes cardinality.) All other entries of the transition matrix are zero.

Thus for each $s \in S$, $P(s, s')$ is nonzero for exactly $2^{|\mathrm{OPP}(s)|}$ states s'. However, given states $s, s' \in S$, it can easily be verified that in $2n$ one-step transitions, s can be transformed into s' with positive probability. Thus the Markov chain with transition matrix P is irreducible. It is also positive recurrent, because the set S is finite. Therefore, as is well known, there is a unique stationary probability distribution for P, i.e., there is a unique vector π on S such that

$$\pi(s) = \sum_{s' \in S} \pi(s') P(s', s) \quad \text{for all } s \in S, \tag{2.2}$$

$$\pi(s) > 0 \quad \text{for all } s \in S, \tag{2.3}$$

and

$$\sum_{s \in S} \pi(s) = 1. \tag{2.4}$$

The stationary probability distribution of P can thus be obtained by first getting a vector that satisfies (2.3) and (2.2) and then normalizing it.

Let γ be the vector on S defined by

$$\gamma(s) = (1-p)^{-|\mathrm{OPP}(s)|}. \tag{2.5}$$

THEOREM 2.1. *The vector γ defined by* (2.5) *satisfies* (2.3) *and* (2.2) *for the matrix P defined by* (2.1).

Proof. Plainly, since $0 < p < 1$, γ satisfies (2.3). To verify (2.2), given $s \in S$, by the definition of P, the summation on the right side of (2.2) need be taken only over those $s' \in S$ such that for some $A \subseteq \mathrm{OPP}(s)$, $s = (s')^A$. In such a case, $A \subseteq \mathrm{POSTOPP}(s)$. Moreover, given $s \in S$ and $A \subseteq \mathrm{POSTOPP}(s)$, there is a unique state s' satisfying $(s')^A = s$. We shall denote this state by s^{-A}. With this notation (2.2) can be rewritten as follows:

$$\gamma(s) = \sum_{A \subseteq \mathrm{POSTOPP}(s)} \gamma(s^{-A}) P(s^{-A}, s) \quad \text{for all } s \in S. \tag{2.6}$$

The proof of (2.6) will use the following two lemmas.

LEMMA 2.2. *If $s \in S$, then*

$$|\mathrm{OPP}(s)| = |\mathrm{POSTOPP}(s)|.$$

Proof. For $j \in \mathrm{OPP}(s)$, let j' be the largest integer such that $s_i = 0$ for $j < i \leq j'$ and let $j'' = j' \bmod n$. It is easy to see that the pairing of j and j'' gives a one-to-one correspondence between $\mathrm{OPP}(s)$ and $\mathrm{POSTOPP}(s)$. □

LEMMA 2.3. *If $s \in S$ and $A \subseteq \mathrm{POSTOPP}(s)$, then*

$$\gamma(s^{-A})\ P(s^{-A}, s) = \left(\frac{p}{1-p}\right)^{|A|}.$$

Proof. Since $(s^{-A})^A = s$ by definition, we have

$$\begin{aligned}\gamma(s^{-A})\ P(s^{-A}, s) &= (1-p)^{-|\mathrm{OPP}(s^{-A})|} \cdot p^{|A|} \cdot (1-p)^{|\mathrm{OPP}(s^{-A})|-|A|} \\ &= \left(\frac{p}{1-p}\right)^{|A|}. \quad \square\end{aligned}$$

We now finish the proof of Theorem 2.1 by verifying (2.6). Note that

$$\begin{aligned}
&\sum_{A\subseteq \mathrm{POSTOPP}(s)} \gamma(s^{-A})\cdot P(s^{-A},s)\\
&= \sum_{A\subseteq \mathrm{POSTOPP}(s)} \left(\frac{p}{1-p}\right)^{|A|} \quad \text{[by Lemma 2.3]}\\
&= \sum_{\ell=0}^{|\mathrm{POSTOPP}(s)|} \binom{|\mathrm{POSTOPP}(s)|}{\ell}\left(\frac{p}{1-p}\right)^{\ell}\\
&= \left(1+\frac{p}{1-p}\right)^{|\mathrm{POSTOPP}(s)|} \quad \text{[by the binomial theorem]}\\
&= (1-p)^{-|\mathrm{POSTOPP}(s)|}\\
&= (1-p)^{-|\mathrm{OPP}(s)|} \quad \text{[by Lemma 2.2].}
\end{aligned}$$

The proof of Theorem 2.1 is thus complete. □

COROLLARY 2.4. *If π is defined by*

$$\pi(s)=\frac{\gamma(s)}{\sum_{s'\in S}\gamma(s')}, \tag{2.7}$$

where γ satisfies (2.5), *then π is the unique stationary probability distribution of the Markov chain with transition probability matrix P.*

3. Average cycle time. A natural measure of the progress of the system is the number of packet movements. In a one-step transition from state s to state s', this is the number of opportunity stations of s that successfully transmit a packet and hence are post-opportunity stations of s', i.e.,

$$|\mathrm{OPP}(s)\cap \mathrm{POSTOPP}(s')|.$$

Of course, such a transition would have positive probability only when $s'=s^A$ for some $A\subseteq \mathrm{OPP}(s)$; and in such a case $|\mathrm{OPP}(s)\cap\mathrm{POSTOPP}(s')|=|A|$. If $s^0,s^1,\dots,s^t,\dots$ denote the sequence of states generated by our Markov chain (s^t is the state at time instant t), the long-run average time per unit progress is

$$\lim_{t\to\infty}\frac{t}{\sum_{i=1}^t|\mathrm{OPP}(s^{i-1})\cap\mathrm{POSTOPP}(s^i)|}. \tag{3.1}$$

Using the strong law of large numbers for Markov chains, one can show (see below) that with probability one,

$$\lim_{t\to\infty}\frac{t}{\sum_{i=1}^t|\mathrm{OPP}(s^{i-1})\cap\mathrm{POSTOPP}(s^i)|}=\frac{1}{p\sum_{s\in S}\pi(s)|\mathrm{OPP}(s)|}. \tag{3.2}$$

Since the long-run average cycle time for a packet, to be denoted by TIME(n,k,p), can be defined to be equal to the long-run average time for nk units progress of the system, we have

$$\mathrm{TIME}(n,k,p)=\frac{nk}{pE(|\mathrm{OPP}|)}, \tag{3.3}$$

where $E(|\mathrm{OPP}|)=\sum_{s\in S}|\mathrm{OPP}(s)|\pi(s)$, the expected number of opportunities under π.

To see how the strong law is used to calculate the limit in (3.1), it helps to consider the associated Markov chain, with state space $S\times S$, which generates the sequence of states

$$(s^0,s^1),(s^1,s^2),\dots,(s^{t-1},s^t),\dots.$$

It is easily verified that this Markov chain has stationary distribution π^* defined by

$$\pi^*(s, s') = \pi(s)P(s, s'),$$

where π and P are defined by (2.7) and (2.1), respectively. By an application of the strong law (Chung [2, Theorem 2, §I.15]), the limit (3.1) can now be seen to equal, with probability one,

$$\frac{1}{\sum_{(s,s')\in S\times S} |\mathrm{OPP}(s) \cap \mathrm{POSTOPP}(s')|\pi^*(s, s')}.$$

This is the same as

$$\begin{aligned}
&\frac{1}{\sum_{s\in S}\sum_{A\subseteq \mathrm{OPP}(s)} |A|\pi(s)P(s, s^A)}\\
&= \frac{1}{\sum_{s\in S}\pi(s)\sum_{\ell=1}^{|\mathrm{OPP}(s)|}\ell\binom{|\mathrm{OPP}(s)|}{\ell}p^{\ell}(1-p)^{|\mathrm{OPP}(s)|-\ell}}\\
&= \frac{1}{\sum_{s\in S}\pi(s)p|\mathrm{OPP}(s)|}\\
&= \frac{1}{p\cdot\sum_{s\in S}\pi(s)|\mathrm{OPP}(s)|},
\end{aligned}$$

where the second-to-last equality holds because the inner sum is a binomial expectation. The verification of (3.2) is now complete.

The following proposition calculates $E(|\mathrm{OPP}|)$ explicitly in terms of n, k, and p.

PROPOSITION 3.1.

$$E(|\mathrm{OPP}|) = \frac{\sum_{\ell=1}^{\min(k,n-k)}\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}{\sum_{\ell=1}^{\min(k,n-k)}\frac{1}{\ell}\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}. \tag{3.4}$$

Proof. Since $k < n$, for every $s \in S$ there must exist j such that $s_j s_{j+1} = 10$. Hence $|\mathrm{OPP}(s)| \geq 1$. Moreover,

$$\mathrm{OPP}(s) \subseteq \{j : 0 \leq j \leq n-1 \text{ and } s_j = 1\},$$

so $|\mathrm{OPP}(s)| \leq k$. Further,

$$\{j+1 : j \in \mathrm{OPP}(s)\} \subseteq \{i : 1 \leq i \leq n \text{ and } s_i = 0\},$$

so $|\mathrm{OPP}(s)| \leq n - k$. We have thus shown that

$$1 \leq |\mathrm{OPP}(s)| \leq \min(k, n-k)$$

for each $s \in S$. For each integer ℓ such that $1 \leq \ell \leq \min(k, n-k)$, let $S_\ell = \{s \in S : |\mathrm{OPP}(s)| = \ell\}$. By Corollary 2.4, we have

$$\begin{aligned}
E(|\mathrm{OPP}|) &= \frac{\sum_{s\in S}|\mathrm{OPP}(s)|\gamma(s)}{\sum_{s\in S}\gamma(s)}\\
&= \frac{\sum_{\ell=1}^{\min(k,n-k)}\ell|S_\ell|(1-p)^{-\ell}}{\sum_{\ell=1}^{\min(k,n-k)}|S_\ell|(1-p)^{-\ell}}.
\end{aligned}$$

The proof will be complete if we can show that

$$|S_\ell| = \frac{n}{\ell}\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1} \quad \text{for } 1 \le \ell \le \min(k, n-k). \tag{3.5}$$

Note that every $s \in S$ is the concatenation of an alternating sequence of nonempty blocks of 0's and nonempty blocks of 1's, so $s \in S_\ell$ if and only if it has one of the following four possible forms:

(i) s is the concatenation of an alternating sequence of ℓ nonempty blocks of 1's and ℓ nonempty blocks of 0's (where s starts with a block of 1's);

(ii) s is the concatenation of an alternating sequence of $\ell + 1$ nonempty blocks of 1's and ℓ nonempty blocks of 0's (where s starts with a block of 1's);

(iii) s is the concatenation of an alternating sequence of ℓ nonempty blocks of 0's and ℓ nonempty blocks of 1's (where s starts with a block of 0's);

(iv) s is the concatenation of an alternating sequence of $\ell + 1$ nonempty blocks of 0's and ℓ nonempty blocks of 1's (where s starts with a block of 0's).

Since any string of length a can be partitioned into b nonempty blocks in $\binom{a-1}{b-1}$ ways and since each $s \in S$ has k 1's and $(n-k)$ 0's, we have

$$\begin{aligned} |S_\ell| &= \binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1} + \binom{k-1}{\ell}\binom{n-k-1}{\ell-1} \\ &\quad + \binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1} + \binom{k-1}{\ell-1}\binom{n-k-1}{\ell} \\ &= \left(1 + \frac{k-\ell}{\ell} + 1 + \frac{n-k-\ell}{\ell}\right)\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1} \\ &= \frac{n}{\ell}\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}. \end{aligned}$$

(Similar counting arguments are discussed in detail in Johnson and Kotz [4], for example.) □

THEOREM 3.2.

$$\text{TIME}(n, k, p) = \frac{nk}{p} \frac{\sum_{\ell=1}^{\min(k,n-k)} \frac{1}{\ell} \cdot \binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}{\sum_{\ell'=1}^{\min(k,n-k)} \binom{k-1}{\ell'-1}\binom{n-k-1}{\ell'-1}(1-p)^{-\ell'}}. \tag{3.6}$$

Proof. This is an immediate consequence of (3.3) and Proposition 3.1. □

4. Optimum throughput. Given a station j, if $P(\text{TRANS at } j)$ denotes the probability, under the stationary distribution π, that a packet gets transmitted to station $(j+1)$, then

$$P(\text{TRANS at } j) = p \cdot \pi\{s \in S : j \in \text{OPP}(s)\}.$$

Since $\pi\{s \in S : j \in \text{OPP}(s)\}$ is independent of j, so is $P(\text{TRANS at } j)$, and we can denote it simply by $P(\text{TRANS})$. It now follows that

$$n \cdot P(\text{TRANS}) = p \sum_{j=0}^{n-1} \pi\{s \in S : j \in \text{OPP}(s)\}.$$

Therefore,

$$n \cdot P(\text{TRANS}) = p \cdot E(|\text{OPP}|). \tag{4.1}$$

The last equality is seen by expressing $|\text{OPP}|$ as a sum of n 0–1 valued random variables. Either of the equivalent expressions in (4.1) can be taken as the definition of the "throughput" of the system.

We now solve the following optimization problem. For fixed n and p we determine the value of k that maximizes $P(\text{TRANS})$. Hence, we determine the value of k that maximizes throughput.

THEOREM 4.1. *Let n, a positive integer, and p, a real number such that $0 < p < 1$, be fixed. Then as a function of k, $P(TRANS)$ is maximized at k_0, where*

$$k_0 = \begin{cases} \frac{n}{2} & \text{if } n \text{ is even,} \\ \frac{n-1}{2} \text{ or } \frac{n+1}{2} & \text{if } n \text{ is odd.} \end{cases} \tag{4.2}$$

The proof of Theorem 4.1 relies on a lemma.

LEMMA 4.2. *Let μ_1 and μ_2 be functions defined on $\{1, \dots, L\}$, where L is a positive integer such that*

(i) $\mu_i(\ell) \geq 0$ *for all ℓ and for $i = 1, 2$;*

(ii) $\mu_i(\ell) = 0$ *implies $\mu_i(\ell') = 0$ for all $\ell' \geq \ell$ and for $i = 1, 2$;*

(iii) $\sum_{\ell=1}^{L} \mu_i(\ell) = 1$, *for $i = 1, 2$; and*

(iv) $\mu_1(\ell + 1)/\mu_1(\ell) \geq \mu_2(\ell + 1)/\mu_2(\ell)$ *for all ℓ such that $\mu_1(\ell) > 0$ and $\mu_2(\ell) > 0$.*

Let φ be a nondecreasing function on $\{1, \dots, L\}$. Then

$$\sum_{\ell=1}^{L} \varphi(\ell)\mu_1(\ell) \geq \sum_{\ell=1}^{L} \varphi(\ell)\mu_2(\ell).$$

[*Remark.* This elementary lemma is a special case of a well-known result in the monotone likelihood ratio theory in statistics (see Lehmann [5, §3.3]). However, for completeness, we include a proof.]

Proof. Let ℓ_0 be the least integer in $\{1, \dots, L\}$ such that $\mu_1(\ell_0) \geq \mu_2(\ell_0)$. Such an integer must exist because of hypothesis (iii). Now hypotheses (iv), (ii), and (i) imply that

$$\mu_1(\ell) \geq \mu_2(\ell) \quad \text{for all } \ell \geq \ell_0. \tag{4.3}$$

If $\ell_0 = 1$, then by (iii) and (4.3), $\mu_1(\ell) = \mu_2(\ell)$ for all ℓ and the assertion of the lemma is true. If $\ell_0 \geq 2$,

$$\begin{aligned}
&\sum_{\ell=1}^{L} \varphi(\ell)[\mu_1(\ell) - \mu_2(\ell)] \\
&= \sum_{\ell=1}^{\ell_0-1} \varphi(\ell)[\mu_1(\ell) - \mu_2(\ell)] + \sum_{\ell=\ell_0}^{L} \varphi(\ell)[\mu_1(\ell) - \mu_2(\ell)] \\
&\geq \varphi(\ell_0 - 1) \cdot \sum_{\ell=1}^{\ell_0-1} [\mu_1(\ell) - \mu_2(\ell)] + \varphi(\ell_0) \sum_{\ell=\ell_0}^{L} [\mu_1(\ell) - \mu_2(\ell)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[since } \varphi \text{ is nondecreasing]} \\
&= [\varphi(\ell_0) - \varphi(\ell_0 - 1)] \sum_{\ell=\ell_0}^{L} [\mu_1(\ell) - \mu_2(\ell)] \quad \text{[by (iii)]} \\
&\geq 0 \quad \text{[by (4.3)]}. \qquad \square
\end{aligned}$$

Proof of Theorem 4.1. Because of (4.1) and (3.4), maximizing $P(\text{TRANS})$ is equivalent to minimizing

$$\sum_{\ell=1}^{\min(k,n-k)} \frac{1}{\ell} \cdot \frac{\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}{\sum_{\ell'=1}^{\min(k,n-k)}\binom{k-1}{\ell'-1}\binom{n-k-1}{\ell'-1}(1-p)^{-\ell'}}.$$

For each positive integer $k < n$, define

$$\mu_k(\ell) = \frac{\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}{\sum_{\ell'=1}^{\min(k,n-k)}\binom{k-1}{\ell'-1}\binom{n-k-1}{\ell'-1}(1-p)^{-\ell'}} \tag{4.4}$$

if $1 \leq \ell \leq \min(k, n-k)$ and define $\mu_k(\ell) = 0$ if $\min(k, n-k) < \ell \leq \min(k_0, n-k_0)$.

Since k_0 satisfies (4.2), it follows that for $k < n$ the functions μ_{k_0} and μ_k satisfy hypotheses (i), (ii), and (iii) of Lemma 4.2. Also,

$$\frac{\mu_k(\ell+1)}{\mu_k(\ell)} = \frac{\binom{k-1}{\ell}\binom{n-k-1}{\ell}(1-p)^{-\ell-1}}{\binom{k-1}{\ell-1}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}} = \frac{k(n-k)-\ell n+\ell^2}{\ell^2}\cdot\frac{1}{1-p}$$

for $1 \leq \ell < \min(k, n-k)$. Therefore,

$$\frac{\mu_{k_0}(\ell+1)}{\mu_{k_0}(\ell)} \geq \frac{\mu_k(\ell+1)}{\mu_k(\ell)}$$

for all $k < n$ and for all ℓ satisfying $1 \leq \ell \leq \min(k, n-k)$. Thus hypothesis (iv) of Lemma 4.2 is also satisfied. So by applying Lemma 4.2 to the nondecreasing function φ defined by $\varphi(\ell) = -\frac{1}{\ell}$, we obtain

$$\sum_{\ell=1}^{\min(k_0,n-k_0)} \frac{1}{\ell}\mu_{k_0}(\ell) \leq \sum_{\ell=1}^{\min(k,n-k)} \frac{1}{\ell}\mu_k(\ell) \quad \text{for all } k < n.$$

This completes the proof of the theorem. □

5. Concluding remarks.

5.1. A bound for TIME(2N,N,1/2). In Berman and Simon [1], the question of obtaining an exact expression for TIME($2N$, N, $1/2$) was raised. Of course, Theorem 3.2 gives the answer. Although the expression (3.6) in Theorem 3.2 is complicated, here is an easy argument to show that

$$\text{TIME}\left(2N, N, \frac{1}{2}\right) \leq \frac{4}{p}N.$$

If μ_p denotes the function defined by the right side of (4.4) when $k = N$ and $n = 2N$ and μ_0 denotes this function when, in addition, $p = 0$, then Lemma 4.2 can be applied routinely to show that

$$\sum_{\ell=1}^{N}\frac{1}{\ell}\mu_p(\ell) \leq \sum_{\ell=1}^{N}\frac{1}{\ell}\mu_0(\ell).$$

Hence

$$
\begin{aligned}
\mathrm{TIME}(2N, N, p) &= \frac{2N^2}{p}\sum_{\ell=1}^{N}\frac{1}{\ell}\mu_p(\ell) \\
&\le \frac{2N^2}{p}\sum_{\ell=1}^{N}\frac{1}{\ell}\mu_0(\ell) \\
&= \frac{2N}{p}\frac{\sum_{\ell=1}^{N}\frac{N}{\ell}\cdot\binom{N-1}{\ell-1}\binom{N-1}{\ell-1}}{\sum_{\ell'=1}^{N}\binom{N-1}{N-\ell'}\binom{N-1}{\ell'-1}} \\
&= \frac{2N}{p}\frac{\sum_{\ell=1}^{N}\binom{N}{N-\ell}\binom{N-1}{\ell-1}}{\sum_{\ell'=1}^{N}\binom{N-1}{N-\ell'}\binom{N-1}{\ell'-1}} \\
&= \frac{2N}{p}\cdot\frac{\binom{2N-1}{N-1}}{\binom{2N-2}{N-1}} \\
&\le \frac{4}{p}N.
\end{aligned}
$$

5.2. Asymptotics. The expression for TIME(n, k, p) given by (3.6) is hard to evaluate if k and $n-k$ are large. We tried approximating it by the technique of moment-generating functions as follows:

Rewrite (3.6) as

$$
\mathrm{TIME}(n, k, p) = \frac{nk}{p}\cdot\frac{\sum_{\ell=1}^{\min(k,n-k)}\binom{k}{k-\ell}\binom{n-k-1}{\ell-1}(1-p)^{-\ell}}{\sum_{\ell'=1}^{\min(k,n-k)}\ell'\binom{k}{k-\ell'}\binom{n-k-1}{\ell'-1}(1-p)^{-\ell'}}. \tag{5.1}
$$

Consider now an urn with k red balls and $(n-k-1)$ black balls. Let X denote the number of black balls drawn when $(k-1)$ balls are drawn from the urn, one by one, without replacement. If $t = (1-p)^{-1}$, we may now write (5.1) as

$$
\begin{aligned}
\mathrm{TIME}(n, k, p) &= \frac{nk}{p}\frac{E(t^{X+1})}{E((X+1)t^{X+1})} \quad \text{[where } E \text{ denotes expectation]} \\
&= \frac{nk}{p}\frac{E(t^{X+1})}{t\cdot\frac{d}{dt}[E(t^{X+1})]}.
\end{aligned}
$$

This last expression can be approximated by regarding the hypergeometric random variable X to be approximately normally distributed. In particular, this technique yields for TIME($2N$, N, p) the approximation

$$
\frac{N}{p}\cdot\frac{1}{\frac{1}{2N}+\frac{1}{4}-\frac{1}{16}\ln(1-p)}.
$$

While this approximation seems to work reasonably well for $p \le 0.5$ as compared to simulated values, it is very poor for values of p close to 1.

It would be desirable to derive a closed-form expression for the limit of

$$\frac{n}{\text{TIME}(n, k, p)}$$

as n approaches ∞, k approaches ∞, and $\frac{k}{n}$ approaches α where $0 < \alpha < 1$. In the particular case where $\alpha = 1/2$, Berman and a referee have conjectured that this limit equals $1 - \sqrt{1 - p}$.

Acknowledgments. Shortly after we obtained our results, L. A. Shepp obtained (3.6) independently. A referee provided a number of detailed comments that led to a substantial change in style as well as shortening of an earlier version.

REFERENCES

[1] P. Berman and J. Simon, *Investigations of fault-tolerant networks of computers* (preliminary version), in Proceedings 20th ACM Symposium on Theory of Computing, Chicago, ACM Press, May 1988, pp. 66–77.

[2] K. L. Chung, *Markov Chains with Stationary Transition Probabilities*, second ed., Springer-Verlag, Berlin 1967.

[3] W. Feller, *An Introduction to Probability Theory and its Applications*, Vol. I, Third ed., Wiley & Sons, New York, 1968.

[4] N. Johnson and S. Kotz, *Urn Models and Their Application*, Wiley and Sons, New York, 1977.

[5] E. Lehmann, *Testing Statistical Hypotheses*, Wiley and Sons, New York, 1959.

[6] A. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 563–572, June 1994

SUBQUADRATIC SIMULATIONS OF BALANCED FORMULAE BY BRANCHING PROGRAMS *

JIN-YI CAI[†] AND RICHARD J. LIPTON[‡]

Abstract. This paper considers Boolean formulae and their simulations by bounded width branching programs. It is shown that every balanced Boolean formula of size s can be simulated by a constant width (width 5) branching program of length $s^{1.811\cdots}$. A lower bound for the translational cost from formulae to permutation branching programs is also presented.

Key words. branching programs, Boolean formulae, Boolean circuits, group theory

AMS subject classifications. 68Q15, 68Q25

1. Introduction.

In a beautiful paper, Barrington [B] showed that the class of languages recognized by (nonuniform) NC^1 circuits (fan-in 2, depth $O(\log n)$ on n inputs) is identical to the class recognized by bounded width branching program with polynomial length. His main motivation was to resolve a conjecture that had been made by Borodin et al. [BDFP]. They had conjectured the opposite is true, namely, that bounded width computations requires exponential length to simulate NC^1; in particular, the majority function requires exponential length.

Barrington showed that any balanced Boolean formula of depth d can be recognized by a branching program of width 5 and length $(2^d)^2$. Thus, if an NC^1 function is represented by a balanced tree of depth d (so the size s of the tree is roughly 2^d), then the length of the branching program will be s^2, quadratic in the size of the tree. (For simplicity, we assume our balanced trees are fully balanced binary trees, as in Barrington's paper. Although our construction works for trees not fully balanced, the estimate of the length is more complicated.) Barrington's work has been generalized by Ben-Or and Cleve [BC] to algebraic formulae. Their simulation, which uses a construction similar to Barrington's, also has a quadratic increase in the length.

The first set of results in this paper is to improve on the upper bound established by Barrington. We show that, for width 5 branching programs, the exponent 2 in Barrington's construction can be improved to 1.811.... Then in §5 we present a lower bound on the translational cost from formulae to branching programs over any finite groups.

Our primary motivation for this work is to study branching programs and the important class NC^1 [Pi][Co]. If we can shed further light on the relationship between branching programs and the class NC^1 in the attempt to lower the exponent of the cost of simulating NC^1 by branching programs, the study will have proved to be worthwhile.

Specifically, we would like to sharpen the estimate of the translational cost from Boolean formulae to branching programs. The class of balanced Boolean formulae of polynomial size is the same as NC^1; thus, this also provides improved simulation of NC^1 by branching programs, although in general, another polynomial factor will appear [Sp], just as in Barrington's simulation. For the class of balanced Boolean formulae, one referee points out that, in fact, people have thought that the quadratic bound of Barrington was optimal, before the current paper. Also, understanding how to encode information into simple groups may help shed light

*Received by the editors May 7, 1990; accepted for publication (in revised form) March 3, 1993. A preliminary version of this paper appeared in the Annual Symposium on Foundations of Computer Science 1989.

[†]Department of Computer Science, Princeton University, Princeton, New Jersey 08544 (jyc@cs.princeton.edu). The work of this author was supported by National Science Foundation grants CGR-8709818 and CCR-9057986.

[‡]Department of Computer Science, Princeton University, Princeton, New Jersey 08544 (rjl@cs.princeton.edu). The work of this author was supported by Defense Advanced Research Projects Agency and Office of Naval Research grants N00014-85-C-0456 and N00014-95-K-0465.

onto why no construction seems possible in solvable groups. These questions are intimately related to constant depth circuits with modulo p gates for a fixed finite set of primes p.

Unlike Barrington's construction, our branching programs in this paper use subprograms that are not restricted to have only two output values for intermediate steps, although the final output is still Boolean. The question of this so-called "weak" versus "strong" representation, namely, whether the less restrictive weak representation (with possibly more than two output values in intermediate steps) is more powerful than strong representation (with exactly two outputs), has been of interest in other upper/lower bound proofs. The present paper provides quantitative evidence that, for branching programs, indeed the less restrictive representation is more powerful.

There already has been some extension of our work since a preliminary version [CL] of this paper appeared in FOCS '89. Cleve [Cl] has proved one of our original conjectures: that indeed the exponent can be reduced to $1 + \varepsilon$ at the cost of increased (depending on ε but still constant) width for the simulating branching program. The width achieved is rather large, and the theoretical question of whether the increase is necessary is still open.

Our lower bound takes a first step toward this direction. Using a Ramsey-type argument, we show that over any finite group there must be a superlinear increase in the length in general. We establish the first superlinear lower bound for a class of branching programs that compute functions with linear formula size.

2. Preliminaries. The branching program model is a generalization of the binary decision tree model. We consider a branching program as given by a leveled directed acyclic graph. It has a "start" node. All links point from nodes of one level to the next, and all nodes of the last level are terminal nodes, labeled *accept* or *reject*. Each nonterminal node is labeled by a Boolean variable and has two links labeled *true* or *false*. A setting of the Boolean variables determines a unique path from the start node to a terminal node. The complexity measures are its width and length (see [B] for details).

We assume our NC^1 circuits are given by balanced binary trees of depth $O(\log n)$. In this paper, we speak of "size" as the tree size, or formula size, and circuits as balanced Boolean formulae. In case the tree is not balanced, one can always balance the tree (at the cost of a polynomial blow up of the circuit size). It is also well known that any NC^1 circuit can be simulated by a Boolean formula of polynomial size. Note that the quadratic cost of Barrington's simulation is also measured in terms of the balanced binary tree size of the circuit. To compare the cost of our simulation with that of Barrington's, we just consider balanced Boolean formulae.

We use some elementary notions and results of group theory; they can be found in any standard textbook in that subject (e.g., [H]).

3. A subquadratic simulation. We first review Barrington's construction [B]. Suppose an AND gate, $f \wedge g$, is given. Let's represent the truth values of f and g (true or false) by some elements of a finite simple group in such a way that the AND gate $f \wedge g$ can be represented likewise. This can be accomplished as follows. Represent "f = true" by some 5-cycle $a \in A_5$ and "f = false" by the identity $1 \in A_5$, where A_n is the alternating group on n letters. Represent g by some $b \in A_5$ and 1 likewise. Call the representation α and β, respectively. Consider the commutator $[\alpha, \beta]$, namely, the representation $\alpha\beta\alpha^{-1}\beta^{-1}$, where say, $\alpha^{-1} = a^{-1}$ or 1 depending on "f = true" or "f = false," respectively. Because the identity 1 commutes with any element of the group, we see immediately that the commutator evaluates to 1 if $f \wedge g$ is false. The commutator construction is completed by choosing a and b carefully so that the commutator $[a, b]$ is yet another 5-cycle. The case with an OR gate is dual. It should be clear how this can be translated to a statement about width 5 branching programs.

We note that by using a commutator at every level of a Boolean formula, the overhead is quadratic, that is, for a Boolean formula of size s, the length of the simulating branching program using this construction has length s^2.

We shall first give some intuitive ideas leading toward a subquadratic simulation of a Boolean formula. Without loss of generality, we assume our Boolean formulae are given by binary trees of depth $O(\log n)$ and the AND and OR gates alternate at every level. We indicate the modifications needed when the Boolean formulae do not conform to the requirements of strict alternation.

Again, assume we are given $f \wedge g$. It is apparent that if we solely depend on commutators, we necessarily end up with a quadratic blowup. On the other hand, with any simulation (without global reconfiguration), the best one can hope for is linear (or close to) linear overhead. If we simply concatenate the branching programs representing f and g, in other words, if we multiply the respective representations instead of forming their commutator, we get the optimal linear overhead. However, over a group, this necessarily will produce multiple representations for $f \wedge g$ for (at least) one of the truth values. This cannot be continued indefinitely, as the sets of representatives for true and false must be kept disjoint. Thus, it appears that certain "cleanup" steps must be taken.

We show that by carefully combining multiplication with commutators, one can simulate a Boolean formula more efficiently, that is, with a simulation of subquadratic length.

Any balanced Boolean formula of depth d can be expressed as $u \wedge (v \vee (x \wedge y))$, $u \vee (v \vee (x \wedge y))$, $u \vee (v \wedge x \wedge y)$, or $u \vee (v \vee x \vee y)$ or their duals, where u is of depth $d-1$, v is of depth $d-2$, and x and y are of depth $d-3$, respectively.

Let $F = u \wedge (v \vee (x \wedge y))$. Let $a = (1\ 4\ 2\ 3\ 5)$, $b = (1\ 2\ 3\ 4\ 5)$, $c = (1\ 4\ 3\ 5\ 2)$, and $d = (1\ 5\ 2\ 4\ 3)$. The following relations can be easily verified:

$$[a, b] = aba^{-1}b^{-1} = (1\ 4\ 2\ 3\ 5)(1\ 2\ 3\ 4\ 5)(5\ 3\ 2\ 4\ 1)(5\ 4\ 3\ 2\ 1) = (1\ 2\ 5\ 3\ 4) = c^{-1},$$

$$[a, b^2] = ab^2a^{-1}b^{-2} = (1\ 4\ 2\ 3\ 5)(1\ 3\ 5\ 2\ 4)(5\ 3\ 2\ 4\ 1)(4\ 2\ 5\ 3\ 1) = (1\ 3\ 2),$$

$$[a, b^2][a, b]^{-1} = [a, b^2]c = (1\ 3\ 2)(1\ 4\ 3\ 5\ 2) = (1\ 5\ 2\ 4\ 3) = d,$$

$$[c, d] = (1\ 4\ 3\ 5\ 2)(1\ 5\ 2\ 4\ 3)(2\ 5\ 3\ 4\ 1)(3\ 4\ 2\ 5\ 1) = (1\ 2\ 4\ 5\ 3).$$

These identities have the following implications. If we represent the truth value of x and y by 1 (if it is true) and b (if it is false) and if we multiply the representations, we get 1 if $x \wedge y$ is true and b or b^2 otherwise. Now let's represent the truth value of v by 1 (if it is true) and a (if it is false) and form the commutator of the representations of v and of $x \wedge y$ and then form the product of this commutator with $[a, b]^{-1} = c = (1\ 4\ 3\ 5\ 2)$. We call this last multiplication a *shift*. As a result, we get c if $v \vee (x \wedge y)$ is true and 1 or d otherwise. Finally, if we represent u by d (if it is true) and 1 (if it is false) and form the commutator of the representations of $v \vee (x \wedge y)$ and u, we end up with a nice clean form: $[c, d] = (1\ 2\ 4\ 5\ 3)$ if $F = u \wedge (v\ \vee (x \wedge y))$ is true and 1 otherwise.

It should be clear that the shift step does not cost anything in the branching program length, as $[a, b] = (1\ 2\ 5\ 3\ 4)$ is a constant and can be absorbed in the previous transition step in the branching program involving a variable.

By recursively applying the construction, we can get the following estimate. Suppose x, y, v, and u respectively represent circuits of depth k, k, $k+1$, and $k+2$ and of size 2^k, 2^k, 2^{k+1}, and 2^{k+2}, and let ℓ_i be the respective lengths of the branching programs. Then we have

$$\ell_{k+3} = 2(\ell_{k+2} + 2(\ell_{k+1} + (\ell_k + \ell_k))).$$

Let $\ell_i = 2^i \ell'_i$, then

$$\ell'_{k+3} = \ell'_{k+2} + \ell'_{k+1} + \ell'_k.$$

We have the characteristic equation

$$\lambda^3 = \lambda^2 + \lambda + 1.$$

By Cardan's formula [J], we can solve this equation exactly, with the only real root

$$\lambda = \frac{1}{3}(1 + (19 + 3\sqrt{33})^{1/3} + (19 - 3\sqrt{33})^{1/3}) \approx 1.839\ldots.$$

Thus, the construction here achieves a subquadratic blowup of $O(s^{1+\log_2 \lambda}) \approx O(s^{1.879\ldots})$.

We also remark that had the function F been $u \vee (v \vee (x \wedge y))$, we would not have to carry out the shift step in the above construction. Notice that if we simply form the commutator of the representations of v and of $x \wedge y$, we get 1 if $v \vee (x \wedge y)$ is true and $[a, b] = (1\ 2\ 5\ 3\ 4)$ or $[a, b^2] = (1\ 3\ 2)$ otherwise. We represent u by 1 or $(1\ 3\ 2)^{-1}(1\ 2\ 5\ 3\ 4) = (1\ 5\ 3\ 2\ 4)$ (in case it is true or false) and form the commutator of the representations of u and of $v \vee (x \wedge y)$, and we have 1 if F is true and

$$[(1\ 5\ 3\ 2\ 4), (1\ 2\ 5\ 3\ 4)] = [(1\ 5\ 3\ 2\ 4), (1\ 3\ 2)] = (1\ 2\ 3\ 4\ 5)$$

if F is false.

Moreover, given $F = u \vee (x \wedge (y \wedge z))$ or $F' = u \vee (x \vee (y \vee z))$, we have the following construction that performs even better. (These 4 cases are exhaustive, using duality.)

Let

$$a = (1\ 3\ 4\ 2\ 5), \qquad a' = (1\ 4\ 2\ 3\ 5), \qquad b = (1\ 4\ 5\ 2\ 3),$$

$$b' = (1\ 2\ 4\ 5\ 3), \qquad c = (1\ 2\ 5\ 3\ 4), \qquad c' = (1\ 3\ 4\ 2\ 5),$$

and

$$\gamma = aa' = bb' = cc' = aba'b' = aca'c' = bcb'c' = (1\ 5\ 4\ 3\ 2),$$

$$\delta = abca'b'c' = (1\ 3\ 4\ 2\ 5).$$

If we consider $a_x b_y c_z a'_x b'_y c'_z$, where $a_x = 1$ if x is true and $a_x = a$ if x is false, similarly for the others, we have:

$$a_x b_y c_z a'_x b'_y c'_z = \begin{cases} 1 & \text{if } x = y = z = 1, \\ \delta = (1\ 3\ 4\ 2\ 5) & \text{if } x = y = z = 0, \\ \gamma = (1\ 5\ 4\ 3\ 2) & \text{otherwise.} \end{cases}$$

For $F = u \vee (x \wedge y \wedge z)$, we represent u by 1 or $\gamma^{-1}\delta = (1\ 5\ 3\ 2\ 4)$ (true or false, respectively), and we have

$$[\gamma, \gamma^{-1}\delta] = [\delta, \gamma^{-1}\delta] = (1\ 2\ 4\ 5\ 3).$$

For $F' = u \vee (x \vee y \vee z)$, we represent u by 1 or γ (true or false, respectively), and we have

$$[\gamma, \delta] = (1\ 4\ 2\ 3\ 5).$$

(This is even slightly better in complexity with a growth of $s^{1.842\cdots}$. We omit the details. For simplicity we do not consider the strict requirements on alternation and multiple fan-in in the remainder of this paper to avoid tedious case analysis. Thus, we assume our tree is balanced and alternate at each level.)

4. Iterated simulation. In this section we indicate an improvement of the construction given in the last section. Again let us look at $F = u \wedge (v \vee (x \wedge y))$. The simple observation is that in the above construction when we have represented $v \vee (x \wedge y)$ by c if it is true and 1 or d if it is false, we can relax somewhat on the representation of u. More specifically, if we have $u = u_1 \vee u_2$, we may represent both u_1 and u_2 by d (true) and 1 (false) and then simply form the multiplication of the representation. This gives us the representation for u as d or d^2 when u is true and 1 otherwise. Now because all powers of d commute, we still get 1 if F is false. However, if F is true we get either

$$[c, d] = (1\ 2\ 4\ 5\ 3) \quad \text{or} \quad [c, d^2] = (2\ 3\ 4).$$

We note that the situation is exactly the same as what we have before in $v \vee (x \wedge y)$. More exactly, after a conjugation, we have the exact dual case as in the representation for $v \vee (x \wedge y)$ just before the shift. (We recall that a shift is free.) Specifically, the pairs a, b and c, d are conjugate to each other by the same conjugation. Let $\alpha = (1\ 4\ 2)$. Conjugation by α gives $c^\alpha = \alpha^{-1} c \alpha = (4\ 2\ 3\ 5\ 1) = a$ and $d^\alpha = (4\ 5\ 1\ 2\ 3) = b$ and thus $[c, d]^\alpha = [a, b]$ and $[c, d^2]^\alpha = [a, b^2]$. (Just like a shift, a conjugation is merely a renaming of the states of the branching program, and thus it is free. But unlike a shift, a conjugation is a simultaneous renaming of the input as well as the output states, and thus in the actual construction of the branching program, the conjugation does not even need to be carried out. We include them in the exposition only to show that inductively the construction can be carried out.)

Of course, now it is natural to iterate on this. Let h_i and h'_i be circuits of depth $k + i$ and $g_{i+1} = h_i \circ h'_i$, where $\circ = \wedge$ if i is even and $\vee$ if i is odd, and $0 \le i \le m - 2$ for some $m \ge 2$. Let f_1 be a circuit of depth $k + 1$ and $f_{i+2} = f_{i+1} \circ g_{i+1}$, where $\circ = \vee$ if i is even and $\wedge$ if i is odd. Finally, let $F_{m+1} = f_m \circ g_m$, where g_m is a circuit of depth $k + m$ and $\circ = \wedge$ if m is even and $\vee$ if m is odd. Our simulation will carry multiple values as we move up the tree. But in the last step we bring it back to the form of single-valued representation in both cases, true or false.

We have the following recurrence relation where ℓ_i denotes the length of a branching program simulating a circuit of depth i and size 2^i.

$$\ell_{k+m+1} = 2(\ell_{k+m} + 2(2\ell_{k+m-2} + \cdots + 2(2\ell_{k+1} + 2(2\ell_k + \ell_{k+1}))\ldots)),$$

or

$$\ell_{k+m+1} = 2^1 \ell_{k+m} + \sum_{i=3}^{m+1} 2^i \ell_{k+m+1-i} + 2^m \ell_{k+1}.$$

Simplifying a bit, let $\ell_i = 2^i \ell'_i$, we have

$$\ell'_{k+m+1} = \ell'_{k+m} + \sum_{i=3}^{m+1} \ell'_{k+m+1-i} + \ell'_{k+1},$$

which has a characteristic equation

$$\chi_{m+1}(\lambda) = \lambda^{m+1} - \left(\lambda^m + \sum_{i=3}^{m+1} \lambda^{m+1-i} + \lambda\right) = 0.$$

LEMMA 4.1. *Each χ_i has a unique positive root $\lambda_i, i \geq 3$. Furthermore, the set $\{\lambda_i\}$ forms a monotonic decreasing sequence with limit*

$$\lambda_* = \frac{1}{3}\left(2 + \left(\frac{25}{2} + \frac{3}{2}\sqrt{69}\right)^{1/3} + \left(\frac{25}{2} - \frac{3}{2}\sqrt{69}\right)^{1/3}\right) \approx 1.7549\ldots.$$

This implies the following theorem.

THEOREM 4.2. *Every balanced Boolean formula of size s can be simulated by a width 5 permutation branching program of length $O(s^{1+\log_2 \lambda_* + o(1)}) \approx O(s^{1.811\ldots})$. Thus, exponent of the simulation is at most 1.811....*

Proof of the lemma. It is easy to verify that $\chi_3(\lambda) = \lambda^3 - \lambda^2 - \lambda - 1$ has a unique real root at $\lambda_3 = \frac{1}{3}(1 + (19 + 3\sqrt{33})^{1/3} + (19 - 3\sqrt{33})^{1/3}) \approx 1.839\ldots$ by Cardan's formula.

Furthermore, χ_3 has a unique local maximum at $-\frac{1}{3}$ of $\chi_3(-\frac{1}{3}) = -\frac{22}{27}$ and a unique local minimum at 1 of $\chi_3(1) = -2$; thus, $\chi_3(\lambda) < 0$ for $\lambda < \lambda_3$ and monotonic increasing for $\lambda > 1$.

Let

$$\lambda_* = \frac{1}{3}\left(2 + \left(\frac{25}{2} + \frac{3}{2}\sqrt{69}\right)^{1/3} + \left(\frac{25}{2} - \frac{3}{2}\sqrt{69}\right)^{1/3}\right) \approx 1.7549\ldots$$

be the unique real root of $\chi_*(\lambda) = (\lambda - 1)^2\lambda - 1$ (Cardan's formulas). Inductively, we assume that χ_m has a unique positive real root λ_m, $\lambda_* < \lambda_m < \cdots < \lambda_3$, and, moreover, χ_m is monotonic increasing for $\lambda > \lambda_*$, and $\chi_m(\lambda) < 0$ for $0 < \lambda < \lambda_m$.

Let

$$\Delta_m(\lambda) = \chi_{m+1}(\lambda) - \chi_m(\lambda) = \lambda^{m-2}((\lambda - 1)^2\lambda - 1) = \lambda^{m-2}\chi_*(\lambda). \tag{1}$$

It follows, by the definition of λ_*, that $\Delta_m(\lambda) > 0$ for $\lambda > \lambda_*$ and $\Delta_m(\lambda) < 0$ for $0 < \lambda < \lambda_*$. Furthermore, it can be verified directly that $\Delta'_m(\lambda) > 0$ for $\lambda > \lambda_*$.

Thus, $\chi_{m+1}(\lambda) = \chi_m(\lambda) + \Delta_m(\lambda) < 0$ for $0 < \lambda \leq \lambda_*$ and monotonic increasing for $\lambda > \lambda_*$; therefore, $\chi_{m+1}(\lambda)$ has a unique positive real root λ_{m+1}, $\lambda_* < \lambda_{m+1} < \lambda_m < \cdots < \lambda_3$. Moreover, $\chi_{m+1}(\lambda) < 0$ for $0 < \lambda < \lambda_{m+1}$. The induction is completed.

It follows that $\lim \lambda_m$ exists. Let $\lambda^* = \lim \lambda_m$. We claim $\lambda^* = \lambda_*$. Suppose not; thus, $\lambda_* < \lambda^*$. By (1), $\chi_m(\lambda_*)$ is some negative constant $-c$ independent of m.

It is easy to show, by directly taking and estimating the derivative on χ_{m+1}, that for all $\lambda > \lambda_*$,

$$\chi'_{m+1}(\lambda) \geq \lambda^m + m\chi_m(\lambda) \geq \lambda_*^m - mc,$$

which tends to infinity as $m \to \infty$.

However, by the intermediate value theorem, for all m, there exists $\tilde{\lambda}_m \geq \lambda_*$, such that the value of $\chi'_m(\lambda)$ at $\tilde{\lambda}_m$ is bounded above:

$$\chi'_m(\tilde{\lambda}_m) = \frac{\chi_m(\lambda_m) - \chi_m(\lambda_*)}{\lambda_m - \lambda_*} \leq \frac{c}{\lambda^* - \lambda_*}.$$

This contradiction shows that $\lim \lambda_m = \lambda_*$. □

5. A superlinear lower bound. We turn our attention to lower bounds in this section. In particular, we ask what is the minimum length of a branching program for functions with linear formula size. In this section we present a lower bound of $\Omega(n \log\log n)$ for the translational cost from NC^1 circuits to permutation branching programs over any finite group. This bound is the first known lower bound for the translational cost.

Several super linear lower bounds are known for branching programs. Chandra, Furst, and Lipton [CFL] showed that the function $\sum_i x_i = n/2$ requires $\Omega(nw(n))$ in the length of any bounded-width branching program, where $w(n)$ is the inverse function of the van der Waerden numbers. Pudlák [Pu] proved an $\Omega(n \log\log n / \log\log\log n)$ lower bound for threshold functions. A lower bound of $\Omega(n \log n)$ for symmetric Boolean functions was achieved by Ajtal et al [Aj]. However, it is not known whether any lower bound applies to functions with linear circuit size. See also [B2] and [BT] for lower bounds for branching programs over specific groups such as S_3 and some solvable groups. Our bound is the first over an arbitrary finite group.

Our lower bound applies only to permutation branching programs and not to (unrestricted) branching programs in general. In fact, we establish the $\Omega(n \log\log n)$ lower bound for the AND function $\bigwedge_{i+1}^{n} x_i$, which has a trivial width two branching program of length n. Our proof is Ramseyian; a similar method has been used in [AM].

THEOREM 5.1. *Any permutation branching program computing the function* AND $\bigwedge_{i=1}^{n} x_i$ *requires length* $\Omega(n \log\log n)$.

Let a permutation branching program over a finite group G be given that computes the logical AND function of n Boolean variables $x_1, x_2, \dots, x_n$.

We assume the branching program has the following normal form

$$BP(x_1, x_2, \dots, x_n) = g_1(x_{i_1}) g_2(x_{i_2}) \dots g_L(x_{i_L}).$$

(See [B2].) Thus, for each step, the transition depends on one Boolean variable x. Furthermore, if x is true, then the transition is identity $g_k(1) = 1$, and if x is false, then the transition is some element of the group G, $g_k(0) = g_k \in G$. Without loss of generality, every permutation branching program can be brought to this form without any increase of length.

We prove a lower bound on the length L of the branching program. Let n_k denote the number of variables that occur exactly k times in the branching program, then $\sum_{k\geq 1} n_k = n$ and $\sum_{k\geq 1} kn_k = L$. If $\sum_{k\geq(\log_3\log_3 n)/2} n_k \geq n/2$, then we are done: $L \geq (n \log\log n)/4$. We assume $\sum_{k<(\log_3\log_3 n)/2} n_k \geq n/2$, and thus there exists $k < (\log_3\log_3 n)/2$, $n_k \geq n/\log_3\log_3 n$. Set all other variables to true, we get a branching program on n_k variables, each variable appears exactly k times, where $k < (\log_3\log_3 n)/2$. Denote $N = n_k$.

Let ℓ_{ij} be the location of the jth appearance of the ith variable x_i. By renaming variables if necessary, we assume that the first appearances are in order, that is,

$$\ell_{11} < \ell_{21} < \cdots < \ell_{N1}.$$

Consider the second appearances of these variables. We select a subset of the variables of cardinality $\geq N^{1/3}$, such that the second appearances of these variables are nicely correlated to the first appearances of the same variables. More precisely, we show that there exists a subsequence $i_1 < i_2 < \cdots < i_m$ of $1, 2, \dots, N$, where $m \geq N^{1/3}$, such that one of the following three alternatives hold:

(1) $\ell_{i_1 2} > \ell_{i_2 2} > \cdots > \ell_{i_m 2}$, or

(2) $\ell_{i_1 2} < \ell_{i_2 2} < \cdots < \ell_{i_m 2}$ and $\ell_{i_1 2} > \ell_{i_m 1}$, or

(3) $\ell_{i_1 1} < \ell_{i_1 2} < \ell_{i_2 1} < \ell_{i_2 2} < \cdots < \ell_{i_m 1} < \ell_{i_m 2}$.

For any sequence of integers, it is well known that we can first obtain either a monotonic decreasing subsequence of length $m' \geq N^{1/3}$ or a monotonic increasing subsequence of length

$m' \geq N^{2/3}$ of the sequence $\ell_{12}, \ell_{22}, \ldots, \ell_{N2}$. If the subsequence is monotonic decreasing, then the first altenative holds.

Suppose it is monotonic increasing. By setting all the other variables to true and by renaming the variables, we may assume the subsequence is $\ell_{12} < \ell_{22} < \cdots < \ell_{m'2}$. For $1 \leq i \leq m'$, let $p(i) = \max\{p \mid \ell_{i2} > \ell_{p1}\}$, that is, the relative place of ℓ_{i2} in the first sequence. Clearly, $p(i) \geq i$ for all i, and $1 \leq p(1) \leq \cdots \leq p(m') = m'$. Now we ask the key question: Is there an i, $1 \leq i \leq m'$, such that $p(i) - i \geq \sqrt{m'}$? If so, then we choose the subset as those with indices between i and $p(i)$:

$$\ell_{i2} < \ell_{i+12} < \cdots < \ell_{p(i)2},$$

and $\ell_{i2} > \ell_{p(i)1}$. Thus, the second alternative holds.

Now suppose the answer is no, that is, for all i, $p(i) - i < \sqrt{m'}$. Then we are going to select our subsequence greedily as follows: Let $q(i) = p(i) + 1$, and

$$i_j = q^{(j-1)}(1), \qquad 1 \leq j \leq \sqrt{m'},$$

where $f^{(k)}$ denotes the kth iterate of a function f. That all i_j, $1 \leq j \leq \sqrt{m'}$, are no greater than m', and thus well defined, is a consequence of our hypothesis for all i, $p(i) - i < \sqrt{m'}$. This implies the third alternative and completes the proof of our claim.

Now one can iterate this process. Suppose s iterations are done, and $N' \geq N^{1/3^s}$ variables remain. Inductively, every successive sequence (consisting of the rth occurrence, $1 < r \leq s$, of the remaining variables) is related to its previous sequences similarly as in that between the first and the second sequence just shown. In fact, we can group together those successive sequences related as in alternative 3, namely

$$\ell_{i_1 r} < \ell_{i_1 r+1} < \cdots < \ell_{i_1 r'} < \ell_{i_2 r} < \ell_{i_2 r+1} < \cdots < \ell_{i_2 r'} < \cdots < \ell_{i_m r} < \ell_{i_m r+1} < \cdots < \ell_{i_m r'},$$

or its reverse,

$$\ell_{i_1 r} > \ell_{i_1 r+1} > \cdots > \ell_{i_1 r'} > \ell_{i_2 r} > \ell_{i_2 r+1} > \cdots > \ell_{i_2 r'} > \cdots > \ell_{i_m r} > \ell_{i_m r+1} > \cdots > \ell_{i_m r'}.$$

Call any maximal such internal $\{j | r \leq j \leq r'\}$ a block.

We focus on the last block, say starting from $t + 1$ to s. By renaming the variables, we assume inductively $x_1, x_2, \ldots, x_{N'}$ are remaining, and

$$\ell_{1,t+1} < \ell_{1,t+2} < \cdots < \ell_{1s} < \ell_{2,t+1} < \ell_{2,t+2} \cdots < \ell_{2s} < \cdots < \ell_{N',t+1} < \ell_{N',t+2} < \cdots < \ell_{N's},$$

and, if $t > 0$, the tth and $t + 1$st sequence are related as in alternative (1) or (2). (The other alternative of reversing all $<$ to $>$ in the above ℓ-sequence is symmetric.)

We will select a subsequence of $\ell_{1s+1}, \ldots, \ell_{N's+1}$, indexed by $i_1 < i_2 < \cdots < i_m$, where $m \geq N'^{1/3}$, such that one of the following three alternatives is true:

(1) $\ell_{i_1 s+1} > \ell_{i_2 s+1} > \cdots > \ell_{i_m s+1}$, or

(2) $\ell_{i_1 s+1} < \ell_{i_2 s+1} < \cdots < \ell_{i_m s+1}$ and $\ell_{i_1 s+1} > \ell_{i_m s}$, or

(3) $\ell_{i_1 t+1} < \ell_{i_1 t+2} < \cdots < \ell_{i_1 s} < \ell_{i_1 s+1} < \ell_{i_2 t+1} < \ell_{i_2 t+2} < \cdots < \ell_{i_2 s} < \ell_{i_2 s+1} < \cdots < \ell_{i_m t+1} < \ell_{i_m t+2} < \cdots < \ell_{i_m s} < \ell_{i_m s+1}$.

The proof is identical to the base case, and we will not repeat it here.

Because $k < (\log_3 \log_3 n)/2 < \log_3 \log_3 N$, the process can be iterated k times. We end up with k sequences each of which has $n' \geq N^{1/3^k}$ variables, and the branching program computes the AND function of these variables (all others are set to true). Moreover, all adjacent blocks of sequences are related in one of three ways as above. Clearly, any block can be collapsed to just one sequence. After the collapse (and renaming the variables) we have $k' \leq k$ sequences, and the branching program looks like

$$S_1 S_2 \dots S_{k'},$$

where each S_j is either

$$g_{1j}(x_1) g_{2j}(x_2) \dots g_{n'j}(x_{n'}),$$

or

$$g_{n'j}(x_{n'}) g_{n'-1j}(x_{n'-1}) \dots g_{1j}(x_1).$$

Consider the map F from $1 \leq i \leq n'$ to $G^{k'}$:

$$i \mapsto \langle s_1(i), s_2(i), \dots, s_{k'}(i) \rangle,$$

where

$$s_j(i) = g_{1j} g_{2j} \cdots g_{ij},$$

in the first case of S_j, or

$$s_j(i) = g_{ij} g_{i-1,j} \cdots g_{1j},$$

in the second case of S_j. (Recall that $g_{\sigma j} = g_{\sigma j}(0) \in G$.)

Because $k < (\log_3 \log_3 n)/2$ and $N \geq \frac{n}{\log_3 \log_3 n}$, it follows easily that[1] $n' \geq N^{1/3^k} > |G|^{k'}$; thus, $F(i) = F(i')$ for some $i < i'$ by the pigeonhole principle. Therefore, for all j, $g_{i+1\,j} \cdots g_{i'j} = 1 \in G$, or $g_{i'j} \cdots g_{i+1\,j} = 1 \in G$, which ever the case may be. This implies that the original branching program evaluates to 1 when all variables (after renaming) between x_{i+1} and $x_{i'}$ are set to false and others set to true. Hence it does not compute the AND function. This completes the proof of our lower bound.

The method we used here to prove our $\Omega(n \log \log n)$ lower bound has been used by Barrington and Straubing to obtain several other lower bounds [BS].

6. Open problems. There are many unanswered questions raised here. We mentioned in the beginning of the paper that Cleve [Cl] has proved the following theorem, which was the first conjecture in the preliminary version of this paper.

THEOREM 6.1 (Cleve). *For any $\epsilon > 0$, an NC^1 circuit of size s can be simulated by a width $2^{2^{O(1/\epsilon)}}$ (permutation) branching program of length $O(s^{1+\epsilon})$.*

In view of this, it is natural to define *Barrington's constants* $\mathcal{B}_k$, for each width k, that is, $\mathcal{B}_k$ is the infimum of $\mathcal{B}$ such that any NC^1 circuit of size s can be simulated by a width k (permutation) branching program of length $O(s^{\mathcal{B}})$. We conjecture that these Barrington's constants are greater than one (and hence nontrivial).

[1] Let $c = |G|$, a constant. Because $k < (\log_3 \log_3 n)/2$, $3^k < \sqrt{\log_3 n}$. As $N \geq \frac{n}{\log_3 \log_3 n}$,

$$\begin{aligned} N \cdot \log_3 \log_3 n &\geq n \\ &\geq c^{\frac{\sqrt{\log_3 n}\, \log_3 \log_3 n}{2}} + \log_c \log_3 \log_3 n \\ &> c^{\frac{\log_3 \log_3 n}{2} \cdot 3^k} \cdot \log_3 \log_3 n. \end{aligned}$$

Thus, $N^{1/3^k} > c^{\frac{\log_3 \log_3 n}{2}} > c^k \geq c^{k'}$, for large n.

CONJECTURE 6.2. *Width k (permutation) branching programs simulating any NC^1 circuit of size s requires length $\Omega(s^{1+\epsilon_k})$, for some $\epsilon_k > 0$.*

Our lower bound in §5 can be viewed as the first step toward settling this conjecture. If this conjecture is true, one may further inquire the exact order of growth of these *Barrington's constants* $\mathcal{B}_k$. The best known bound for $\mathcal{B}_5$ is 1.811 .... It is not clear what to expect in general. A tight simulation of circuits by branching programs could offer the possibility of proving lower bounds for circuits size, whereas a reasonable width could be valuable in hardware design pertaining reconfigurable chips.

Acknowledgments. We thank Sandeep Bhatt, Walter Feit, Mike Fischer, Merrick Furst, Roger Howe, Herb Scarf, and George Seligman for helpful conversations. We thank the two anonymous referees for many comments.

REFERENCES

[Aj] M. AJTAL, L. BABAI, P. HAJNAL, J. KOMLÓS, P. PUDLÁK, V. RÖDL, E. SZEMERÉDI, AND G. TURÁN, *Two lower bounds for branching programs*, in Proc. 18th ACM STOC, Berkeley, CA, 1986, pp. 30–38.

[AM] N. ALON AND W. MAASS, *Meanders and their applications in lower bounds arguments*, JCSS, 38 (1988), pp. 118–129.

[B] D. BARRINGTON, *Bounded-width polynomial-size branching programs recognizes exactly those languages* in NC^1, JCSS, 38 (1990) pp. 150–324.

[B2] ———, *Width-3 permutation branching programs*, Tech. memorandum TM-291, MIT Laboratory for Computer Science, Cambridge, MA, 1985.

[BS] D. BARRINGTON AND H. STRAUBING, *Superlinear lower bounds for bounded-width branching programs*, in Proc. 6th Structure in Complexity Theory Conference, IEEE Computer Society Press, Alamitos, CA, 1991, pp. 305–313.

[BT] D. BARRINGTON AND D. THÉRIEN, *Non-uniform automata over groups*, Lecture Notes in Comp. Sci., Springer-Verlag, 267, 1987, pp. 163–173.

[BC] M. BEN-OR AND R. CLEVE, *Computing algebraic formulas using a constant number of registers*, in Proc. 20th ACM STOC, Chicago, Illinois, 1988, pp. 254–257.

[BDFP] A. BORODIN, D. DOLEV, F. E. FICH, AND W. PAUL, *Bounds for width-2 branching programs*, SIAM J. Comput., 15 (1986), pp. 549–560.

[CL] J. CAI AND R. LIPTON, *Subquadratic simulations of circuits by branching programs*, in Proc. 30th IEEE FOCS IEEE Computer Society Press, Alamitos, CA, 1989, pp. 568–573.

[CFL] A. CHANDRA, M. FURST, AND R. J. LIPTON, *Multiparty protocols*, in Proc. 15th ACM STOC, the ACM Inc., 11 West 42nd Street, New York, 1983, pp. 94–99.

[Cl] R. CLEVE, *Towards optimal simulations of formulas by bounded-width programs*, in Proc. STOC, 22 (1990), Baltimore, MD, pp. 271–277.

[Co] S. COOK, *The taxonomy of problems with fast parallel algorithms*, Inform. and Control (Shenyang), 64 (1985), pp. 2–22.

[H] M. HALL, *The Theory of Groups*, MacMillan, New York, 1959.

[J] N. JACOBSON, *Basic Algebra*, Vol 1, W. H. Freeman and Company, New York, 1985.

[Pi] N. PIPPENGER, *On simultaneous resource bounds (preliminary version)*, in Proc. 20th IEEE FOCS, IEEE Computer Society Press, Alamitos, CA, 1979, pp. 307–311.

[Pu] P. PUDLÁK, *A lower bound on complexity of branching programs*, 11th MFCS, Lecture Notes in Comput. Sci., Springer-Verlag, vol. 176, pp. 480–489.

[Sp] P. SPIRA, *On time-hardware complexity tradeoffs for Boolean functions*, in Proc. 4th Hawaii Symposium on System Sciences, North Hollywood, CA, Western Periodicals Co., 1971, pp. 525–527.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 573–597, June 1994

LINEAR TIME ALGORITHMS AND NP-COMPLETE PROBLEMS*

ETIENNE GRANDJEAN†

Abstract. This paper defines and studies a computational model (a random access machine with powerful input/output instructions), and shows that the classes DLINEAR and NLINEAR of problems computable in deterministic (respectively, nondeterministic) linear time in this model of computation are robust and powerful. In particular, DLINEAR includes most of the concrete problems commonly regarded as computable in linear time (such as graph problems, topological sorting, strong connectivity, etc.). Most combinatorial NP-complete problems are in NLINEAR. The interest of NLINEAR class is enhanced by the fact that some natural NP-complete problems, for example, "reduction of incompletely specified automata" (RISA), are NLINEAR-complete (consequently, NLINEAR $\neq$ DLINEAR if and only if RISA $\notin$ DLINEAR). This notion strengthens NP-completeness, as this paper argues that propositional satisfiability is not NLINEAR complete.

Key words. linear time, model of computation, random access machine, input/output process, uniform cost measure, Turing machine, simulation, algorithms on graphs, sorting, reduction, NP-completeness, determinism, nondeterminism

AMS subject classifications. 68Q, 68P, 68R, 03D, 05C

1. Introduction and discussion.

It is usually admitted that the most efficient algorithms run in linear time. However, as far as we know, there is no robust and canonical definition of linear time in literature. (On the contrary, polynomial time is a very robust notion: it is sensitive neither to machine model nor to presentation of the input, provided they are "reasonable.") Papers and books (e.g., [AHU1], [AHU2], [FGS], [Kn], [Me], [Se]) concerning algorithms explicitly or, more often, implicitly use a variant of random access machine (RAM of Cook and Reckhow [CoRe]) but do not precisely explain or do not justify what "the good model" should be: what kind of elementary instructions must be taken (addition, subtraction, multiplication)?, what is the input/output device? (A notable exception is paper [AnVa] that describes and justifies the random access computer (RAC) model; however, their study is partial.) One generally adopts the uniform time criterion: each instruction requires one time unit, but it is not clear what an elementary instruction should be.

An algorithm runs in linear time if it performs $O(n)$ instructions on each input of "length" (or "size") n. Because the notion of instruction depends on the chosen machine, the notion of input length depends on the nature of the input. The length of a word $w \in \sum^n$ in a finite alphabet $\sum$ is n. The size of a graph (represented by its list of edges or by the successor lists of its vertices) is $m + p$, where m (respectively, p) is the number of vertices (respectively, edges). For example, [Ta] and [AHU1] compute the strongly connected components of a directed graph in "linear time" $O(m + p)$. The length of a propositional formula is often (but not always) the number n of its symbols: occurrences of connectives, parentheses, and variables (e.g., the length of formula $(p_0 \bigvee \lrcorner p_1) \bigwedge p_{10}$ is 8). [DoGa], [ItMa1], [ItMa2], and [Mi] describe some decision algorithms for satisfiability of Horn propositional formulas whose runtime is linear $O(n)$.

Gurevich and Shelah [GuSh] and Graedel [Gl] gave two "robust closures" of linear time. Previously, Schnorr [Sr] had similarly defined "quasi-linear time"(i.e., time $O(n(\log n)^{O(1)})$ for some Turing machine) and proved that many NP-complete problems belong to nondeterministic quasi-linear time class and are complete for this class. Those authors define extensions of linear time because, as [GuSh] explains, "It is possible that there is no universal notion of linear time and different versions of linear time are appropriate to different applications."

*Received by the editors December 9, 1991; accepted for publication (in revised form) March 8, 1993.

†Université de Caen, Laboratoire d'Algorithmique et d'Intelligence Artificielle de Caen, 14032 Caen Cedex, France (grandjean@univ-caen.fr).

The present paper adopts the opposite point of view. We define and justify a unified, robust, and powerful notion of linear time both in deterministic and nondeterministic cases. In a former paper [Gr3], we define a complexity class, named LINEAR, by using a variant of classical RAM of [CoRe] under logarithmic cost criterion (time of each instruction is the total number of bits it manipulates, i.e., the sum of lengths of the integers it involves). Our RAM extends the classical model in some points and restricts it in some other ones. (Our purpose is to get a more realistic and more robust machine.) Let us describe informally the differences.

The input process of classical RAM is very restrictive: it can only read one bit at a time. (The output instruction of [CoRe] is more powerful: the RAM writes the whole contents of a register onto the output tape.) Schoenhage [Sc2] has proved that this RAM cannot compute the very simple function $w \mapsto ww$ (repetition of the input word w!) in linear time under logarithmic cost. We allow our RAM to *read its input in blocks* by an instruction, denoted $\text{READ}_v(u)$, which stores in register u a subword of the input as long as the contents of register v. Note that our input and output instructions are symmetrical to each other (in some manner).

In the classical model, the RAM memory can be used in a very *scattered* way because addresses and register contents are unbounded. Our RAM is similar to the RAC of Angluin and Valiant [AnVa], [GuSh]: a computation is allowed to use only integers *polynomially bounded* in the input length n (i.e., their length is $O(\log n)$).

The only available operations of the RAM of [CoRe] are addition (+) and subtraction (−); our RAM can perform any operation computable in linear time on a Turing machine (i.e., intuitively, "easy" operation): +, −, and concatenation but neither multiplication nor division. (Invariance properties of RAMs with compact memory and linear time Turing computable operations have been stated in [GrRo].)

Let LINEAR be the class of functions computable in linear time on the RAM described above. More precisely, a function $f : \sum^* \to \sum^*$ is in LINEAR if and only if there is such a RAM that computes (for each input $w \in \sum^*$) the value $f(w)$ in time $O(\text{length}(w))$ under logarithmic cost. In [Gr3] we have shown that LINEAR is a robust complexity class. However, we are unable to prove that it is closed under composition (i.e., if $f, g \in$ LINEAR, then $fog \in$ LINEAR) unless RAMs can use files (to contain intermediate results).

In this paper we still study the RAM as defined above but which uses the *uniform cost criterion*. This does not change the invariance properties of the machine model but we prove that the class, denoted DLINEAR, of functions computable by RAMs *without files* within time $O(n/\log n)$ (under uniform cost: n is the length of the input) is closed under composition. Note that DLINEAR $\subseteq$ LINEAR. The $O(n/\log n)$ time bound of DLINEAR seems to be very restrictive but is justified by the following facts:

(i) Katajainen et al. [KvLP] have proved that any function f computable in linear time $O(n)$ on a Turing machine can be computed in time $O(n \log\log n)$ on the classical RAM of [CoRe] (under logarithmic cost), and they have mentioned that "with better input/output pattern the simulation could perhaps be sped up further." We can prove (cf. §3 below) that function f is in DLINEAR (i.e., is computable by our RAM in time $O(n/\log n)$ under uniform cost).

(ii) Many (most?) concrete problems commonly regarded as computable in linear time belong to DLINEAR; that contrasts with the (abovementioned) lower bound result of Schönhage and shows the power of our input pattern.

The rest of the present paper is dedicated to the nondeterministic version of linear time. As mentioned above, Schnorr [Sr] has proved that many NP-complete problems are complete in nondeterministic quasi-linear time class (via deterministic quasi-linear time reduction). His computational model was Turing machine. We improve his results as follows. Let NLINEAR denote the nondeterministic version of the DLINEAR class; the RAM has a nondeterministic instruction because it can "guess" an integer. Note the inclusions NTIME $(n) \subseteq$ NLINEAR $\subseteq$ NTIME$(n \log n)$.

We show that many combinatorial NP-complete problems (20 among the 21 problems listed by Karp [Ka]) belong to NLINEAR. Moreover, we prove that *some* NP-complete problems (e.g., reduction of incompletely specified automata or RISA) are complete in NLINEAR via DLINEAR reductions (more strongly, reductions are computable in linear time on Turing machines). That improves the results of [Gr1], [Gr2], [Ra1], [Ra2], which state that each problem in NTIME(n) is reducible to RISA (and to some other NP-complete problems) in deterministic linear time. The proof of the strengthened result is similar to the original proof. That entails the following equivalence

$$\text{RISA} \notin \text{DLINEAR iff DLINEAR} \neq \text{NLINEAR},$$

which can be compared with the well-known equivalence

$$\text{SAT} \notin \text{P iff P} \neq \text{NP}.$$

Let us mention some related results. To our knowledge, Dewdney [De] was the first author who defined and exhibited linear time reductions. In particular, he proved that several NP-complete problems, including 3-SAT (i.e., satisfiability for clauses with three literals) and 3-COLORABILITY are mutually reducible via linear time reductions (on Turing machines). He assumes that inputs are previously normalized, for example, the set of variables that occur in a propositional formula must be exactly $\{p_1, p_2, \ldots, p_m\}$. More recently, Hunt and Stearns [HuSt] proved that a number of problems, including SAT, are mutually reducible via some comparable but weaker reduction (linear length bounded and computable in quasi-linear time). Note that if two problems are reducible to each other in linear time (e.g., two NLINEAR-complete problems), then they have the same time complexity.

After our machine model is defined in §2, the deterministic and nondeterministic linear time classes defined (DLINEAR and NLINEAR) are shown to be very robust (§3). In §4 we exhibit many problems in DLINEAR that are useful in the proof that most concrete problems usually regarded as computable in linear time belong to DLINEAR. Section 5 states properties of linear time reductions. In §6 we show that most combinatorial NP problems belong to NLINEAR. In §7 we prove that several NP-complete problems (e.g., RISA) are NLINEAR-complete, but we argue that propositional satisfiability must not be NLINEAR-complete. We also state other characterizations of the NLINEAR class. Section 8 gives conclusions and open problems.

2. Preliminaries: definitions of our machine model and of linear time complexity. Let $\log n$ denote the logarithm of n in base 2.

Let $\Gamma = \{1, 2, \ldots, d\}$ be a fixed alphabet, with $d \geq 2$. For convenience we identify a word $w = w_0 w_1 \ldots w_{n-1} \in \Gamma^n$ with the integer it represents in d-adic notation, that is,

$$w = \sum_{i<n} w_i d^i.$$

(In particular, the empty word is identified with zero.) Let length$(w) = n$; in particular, length$(0) = 0$.

Let c, d be some integers and let $\Gamma, \sum$ be alphabets such that $\Gamma = \{1, 2, \ldots, d\}$ and $\sum \subseteq \Gamma$. ($\sum$ will be called the input/output alphabet. Sometimes we will use more usual alphabets, including, for example, parentheses.) Our machine model, named DRAM, is the deterministic random access machine of [CoRe] (its sequence of registers are denoted $R(0)$, $R(1)$, $R(2)$, ...) with the following changes.

(i) An input (respectively, output) is a word $w \in \sum^n$ (respectively, $v \in \sum^m$) that is contained (respectively, that will be written) in a special one-way read-only (respectively, write-only) tape called the input (respectively, output) tape.

(ii) The contents of each register $R(i)$ is a nonnegative integer (which is initially 0) whose length, in d-adic notation, is at most $c \log n$ (similar to the memory condition of the RAC in [AnVa]); consequently, the total number of registers used by a DRAM is polynomially bounded in n.

(iii) A DRAM uses a fixed set of operations. All of them have to be computable in linear time on a deterministic Turing machine (e.g., $+$, $-$, concatenation, shift, length), denoted "linear time Turing computable" (LTTC).

(iv) The input/output instructions of a DRAM are the following: WRITE$(R(i))$ writes the contents of register $R(i)$ onto the output tape. READ$_{R(i)}(R(j))$ stores in register $R(j)$ a portion of the input of length L, where L is the length of the integer that register $R(i)$ contains at this moment (a special case occurs: if the nonread part of the input has length less than L, then the whole remainder is stored in $R(j)$).

An NRAM is the nondeterministic version of the previous DRAM. It works similarly but has the additional ability of guessing an integer by the nondeterministic instruction GUESS$(R(i))$ that stores any integer of length at most $c \log n$ in register $R(i)$.

We are now ready to define our time complexity classes.

DEFINITION. *Let $T(n) \geq n/\log n$ be any time function. A function $f : \sum^* \to \sum^*$ (respectively, a language $A \subseteq \sum^*$) belongs to class* DRAM $(T(n))$ *is f is computable (respectively, A is recognizable) by a* DRAM *that executes $O(T(n))$ instructions (uniform time criterion), where n denotes the length of the input w ($w \in \sum^n$). The class of languages* NRAM $(T(n))$ *is defined similarly.*

Convention. In this paper, when we write that a word w of length n is *accepted* by a nondeterministic machine *within some resource* (time $O(T(n))$, space $O(S(n)), \ldots$), we mean that there is an accepting computation of the machine (on that input w) that respects that resource (time $O(T(n)), \ldots$).

Notation. DLINEAR = DRAM$(n/\log n)$; NLINEAR = NRAM$(n/\log n)$.

Remark. These notations are justified by the fact that the time required by a DRAM$(n/\log n)$ (respectively, NRAM$(n/\log n)$) computation is linear (i.e., $O(n)$) under logarithmic cost.

Comments. The definition of the DLINEAR class has two features that at first seem counterintuitive.

(1) The seeming combination of Turing machine features (LTTC operations) with random access. Proposition 3.1(vi) below shows that such a machine can be simulated in linear time by a classical RAM (of [CoRe]) using addition and subtraction (and no other LTTC operation) and shows the robustness of our computational model. Notice that allowing LTTC operations is useful to prove that concrete problems (e.g., NORMALIZE and HORN-SAT; see §4) are in DLINEAR.

(2) The use of the strange time bound $O(n/\log n)$ rather than $O(n)$. This is justified by the following arguments.

(a) It is reasonable to require the available memory to be polynomial, implying that the maximal available address is of the form $N = \Theta(n^d)$, for a fixed d.

(b) In a real computer, addressing a big address register seems to require the same time as addressing a low address one; that explains why the uniform cost criterion is chosen.

(c) For an input of length n in a *fixed finite* alphabet, it is not reasonable to define a linear time computation as a sequence of $O(n)$ instructions because one instruction *does not* involve a *fixed number* of bits. We believe that linear time must be at most $O(n)$ under logarithmic cost measure. As argued above, any instruction is charged $\Theta(\log N) = \Theta(\log n)$ for that measure; that entails choosing the uniform time bound $O(n/\log n)$.

Conventions. In the sequel, in each case when the time cost criterion is not mentioned, we shall implicitly assume that the RAM uses *uniform cost criterion.* Letter n will always denote the *length* of the input.

3. Classes DLINEAR and NLINEAR are robust. On one hand, Katajainen et al. [KvLP] have shown that any function computable in linear time $O(n)$ on a Turing machine can be computed in time $O(n \log\log n)$ on an ordinary RAM (of [CoRe]) under logarithmic cost. (Recall that in one instruction, such a RAM can read at most one bit of the input.) On the other hand, Schoenhage [Sc2] has proved that time bound $O(n \log\log n)$ cannot be replaced by $O(n)$. In this section we show that the impossibility comes from the weakness of the input/output process of usual RAM and we state several robustness properties of the DRAM (respectively, NRAM).

For technical reasons, it is useful to define a variant of DRAM *without* input/output instructions. Instead, the input is divided and stored in a standard way at the beginning of a computation. (That can be compared with the start condition of RACs in [AnVa]; however, that paper does not state precisely how the input is stored in memory.)

Notation. Let $k \geq 1$ be a fixed integer. Let $b_k(n) = \ulcorner 1/k \log(n+1) \urcorner$.

DEFINITION. *Let $k \geq 1$ be a fixed integer. A* DRAM$_k$ *is like a* DRAM *except for input/output.*

(i) *Each input word $w \in \sum^n$ of length n is divided into nonempty words $w_1, w_2, \ldots, w_q$ initially stored in respective registers $R(1), R(2), \ldots, R(q)$:*

$$w = w_1 \,\hat{}\, w_2 \ldots \hat{}\, w_q (\text{concatenation}),$$

where length $(w_i) = b_k(n)$ for $i < q$ and $0 <$ length $(w_q) \leq b_k(n)$ (the other registers $R(q+1), \ldots$ initially contain zero). $w_1, w_2, \ldots, w_q$ are called the $b_k(n)$-blocks of w.

(ii) *An output word v (if any) is similarly divided into nonempty $b_k(n)$-blocks $v_1, v_2, \ldots, v_s$ (the last block v_s may be shorter), respectively, stored in registers $R(1), R(2), \ldots, R(s)$ at the end of a computation (with $R(s+1) = 0$). Note that n is not the length of the output but is still the length of the input. We define an* NRAM$_k$ *in the same manner.*

DEFINITION. *A function (respectively, a language) belongs to* DRAM$_k(n/\log n)$ *(respectively,* NRAM$_k(n/\log n)$*) if it is computed (respectively, recognized) by a* DRAM$_k$ *(respectively,* NRAM$_k$*) in time $O(n/\log n)$ under uniform cost.*

Remark. The important things about the block length $b_k(n)$ is the fact that it is easy to compute and that $b_k(n) = \Theta(\log n)$ for $n \geq 1$ and then the number q (respectively, s) of blocks of the input (respectively, output) is $O(n/\log n)$. We shall also use the fact that $\lim_{k \to +\infty} b_k(n) = 0$. The following proposition shows how robust the classes DLINEAR = DRAM$(n/\log n)$ and NLINEAR = NRAM$(n/\log n)$ are.

PROPOSITION 3.1. *Let $f : \sum^* \to \sum^*$ be a function.*

(i) *For all integers k, k', $f \in$* DRAM$_k(n/\log n)$ *if and only if $f \in$* DRAM$_{k'}(n/\log n)$.

(ii) *$f \in$* DRAM$(n/\log n)$ *if and only if $f \in$* DRAM$_k(n/\log n)$ *for some k.*

(iii) *If f is computable in linear time $O(n)$ by a deterministic Turing machine, then $f \in$* DRAM$(n/\log n)$.

(iv) *Assertions* (i)–(iii) *hold for languages (instead of functions) and the nondeterministic classes* NRAM$(n/\log n)$ *and* NRAM$_k(n/\log n)$.

(v) *Let $f, g : \sum^* \to \sum^*$ be two functions such that $f, g \in$* DRAM$(n/\log n)$; *then $g \circ f \in$* DRAM$(n/\log n)$.

(vi) *Let $f \in$* DRAM$(n/\log n)$ *and $\epsilon > 0$ any real number, then there is an integer K such that for each $k \geq K$, there are a* DRAM$_k$ *and a* DRAM *that compute f in time $O(n/\log n)$ and use exclusively the operation $+$ (respectively, concatentation) and integers $O(n^{1+\epsilon})$.*

(vii) *Let* $A \in \mathrm{NRAM}(n/\log n)$, *then there is an integer* K *such that for each* $k \geq K$, *there are an* NRAM_k *and an* NRAM *that recognize* A *in time* $O(n/\log n)$ *and use exclusively the successor operation* (*i.e., function* $x \mapsto x+1$) *or concatentation and integers* $O(n/\log n)$.

Sketch of proof. (i) and (ii) are roughly proved as follows: the simulating machine reorganizes blocks of the input (respectively, output) by using new LTTC operations (e.g., the length function).

To prove (i), it is sufficient to demonstrate

$$\text{(i}')\qquad f \in \mathrm{DRAM}_k(n/\log n) \text{ iff } f \in \mathrm{DRAM}_{k+1}(n/\log n).$$

To prove (i′) note that $b_{k+1}(n) \leq b_k(n) \leq 2b_{k+1}(n)$ (since $1/(k+1) \leq \frac{1}{k} \leq 2/(k+1)$, and thus an input organized in $b_{k+1}(n)$-blocks is reorganized into $b_k(n)$-blocks by concatenating fragments of two consecutive $b_{k+1}(n)$-blocks to obtain each $b_k(n)$-block. Note that values n, $b_k(n)$, and $b_{k+1}(n)$ have to be initially computed.

To prove (ii), use the fact that (by definition) a DRAM performs only $O(n/\log n)$ read (respectively, write) instructions on integers of length $O(\log n)$; for example, a read instruction is simulated on a DRAM_k by the concatenation of $O(1)$ (fragments of) $b_k(n)$-blocks. This proves the implication $f \in \mathrm{DRAM}(n/\log n)$ implies $f \in \mathrm{DRAM}_k(n/\log n)$. Let us prove the converse. The simulating DRAM first reads the input by blocks by the following subroutine (it stores into registers $R(1), R(2), \ldots, R(m)$ the respective subwords $w_1, w_2, \ldots, w_m$ of the input word w in such a way that $w = w_1 \,\hat{}\, w_2 \,\hat{}\, \ldots w_m$, where $\mathrm{length}(w_i) = \mathrm{length}(i)$ for $i < m$ and $0 < \mathrm{length}(w_m) \leq \mathrm{length}(m)$):

```
begin
    $R(0) \leftarrow 1$
    while the input is not entirely stored do
        begin
            $\mathrm{READ}_{R(0)}(R(R(0)))$
            $R(0) \leftarrow R(0) + 1$
        end
end;
```

We easily check that $m = \Theta(n/\log n)$. Then the simulating machine reorganizes the stored input into $b_k(n)$-blocks as above. This proves (ii).

(iii) easily follows from (ii) and the classical techniques of blocks (see for example [HPV] and [KvLP]) represent each block of $b_k(n)$ consecutive cells of a tape (of the Turing machine) by one register of a DRAM_k.

(iv) is proved exactly as (i)–(iii).

To prove (v), use (ii) and distinguish two cases according to the length, denoted L, of the intermediate value $f(w)$.

(1) $n^{1/2} \leq L = O(n)$, where $n = \mathrm{length}(w)$). It is easy to convert the $b_k(n)$-blocks of $f(w)$ into $b_k(L)$-blocks and to convert the $b_k(L)$-blocks of $g\ o\ f(w)$ into $b_k(n)$-blocks (because $\log L = \Theta(\log n)$).

(2) $L < n^{1/2}$. Block conversion of words $f(w)$ and $g\ o\ f(w)$ is easy because they have only $O(n^{1/2})$ blocks of length $O(\log n)$.

If we use (ii), assertion (vi) is essentially the same as Corollary 3.3 in [Gr3] with uniform time cost instead of logarithmic time cost; moreover, the proof can be simplified (because we no longer need to distinguish "little addresses"). It essentially combines two techniques: division of registers into "small registers" of length $b_k(n)$, and hashing techniques (and pairing functions) to encode big addresses with "small addresses" $O(n^{1+\epsilon})$. More precisely, a big address u is first divided into b-blocks (i.e., blocks of length $b = b_k(n)$): $u = u_m \ldots u_1 u_0$

(concatenation). Then we successively construct the numbers $p_1 = \text{PAIR}(u_0, u_1)$, $p_2 = \text{PAIR}(p_1, u_2), \ldots, p_m = \text{PAIR}(p_{m-1}, u_m)$. The big address u is roughly "simulated" by p_m, which is $O(n)$. (PAIR is a dynamical pairing function whose values are $O(n)$. It can be stored in a two-dimensional array whose first index is $O(n)$ and whose second index is a b-block, and then is $O(2^b) = O(n^{\epsilon/2})$. It can be simulated by a one-dimensional array of indices $O(n^{1+\epsilon})$. For more details, see [Gr3] or see below the proof of Proposition 4.1 where these techniques are reused.) Note that several LTTC operations on small operands are used. These operations can be initially precomputed (for all possible small operands) and stored in several arrays by only using addition (respectively, concatenation). This proves (vi) for a DRAM_k. The same result (vi) is obtained for a DRAM by using the following trick: for a fixed (sufficiently large) k, compute successively all the k-tuples $u_1, u_2, \ldots, u_k$ of integers such that $u_i \leq u_k = j$ for all $i < k$, for successive integers $j = 1, 2, 3, \ldots$, and for each produced k-tuple, read a part of the input of length equal to $\text{length}(u_k)$. (Note: this process generalizes the one presented above to prove (ii).) It is easy but tedious to show that length (u_k) is always less than about $1/k$ length(n) and that the input is entirely read in time $O(n/\log n)$.

(vii) is proved by variants of the above techniques (simplified): registers are divided into "little registers" of length $b_k(n)$ and addressing is simulated by an essential use of nondeterminism as described below. Let $a_1 < a_2 \ldots < a_m$ denote the list of distinct addresses (in increasing order) used by the simulated NRAM_k denoted M. First, the simulating NRAM_k M' guesses that list. Note that $\sum_{i \leq m}$ length $(a_i) \leq n$ and then $m = O(n/\log n)$. The main idea of the simulation is to represent the address a_i register (of M) by a register (of M') of address i. M' has two one-dimensional arrays, denoted $\text{ADDRESS}(1, \ldots, m)$ and $\text{CONTENTS}(1, \ldots, m)$, such that at each moment of the simulation $\text{ADDRESS}(i) = a_i$ $(1 \leq i \leq m)$ and $\text{CONTENTS}(i) =$ contents of register with address a_i (in M). (M' is a "multimemory" NRAM_k.) An access (by M) to some register of address α is simulated by M' as follows: guess an integer $i \in \{1, 2, \ldots, m\}$; check if $\text{ADDRESS}(i)$ (that is a_i) is equal to α (otherwise reject); and use $\text{CONTENTS}(i)$. Note that our multimemory NRAM_k M' can be simulated in linear time by an ordinary NRAM_k (interleave the arrays; see [CoRe] for details) and that we exclusively use addresses $O(m) = O(n/\log n)$. Notice that a NRAM using only the successor operation can simulate this NRAM_k: it initially guesses n and computes $b = b_k(n)$ and a block of length b. □

Remark. The exact function $b_k(n)$ we choose (for block length) is not essential. Proposition 3.1 and its proof still hold for other "easily computable" functions $\Theta(\log n)$; for example, $\llcorner 1/k \log n \lrcorner$ (with $n \geq 2^k$) or $\ulcorner 1/k \text{ length}(n) \urcorner$.

Equality $\text{DLINEAR} = \text{DRAM}_k(n/\log n)$ is essential in our proof that DLINEAR class is closed under composition. That characterization (respectively, the similar one for NLINEAR) will be useful in proofs that concrete problems belong to DLINEAR (respectively, NLINEAR).

4. Many problems computable in linear time. In this section we prove that most problems usually regarded as computable in linear time belong to DLINEAR. In the proofs we need several standard DLINEAR subroutines, for example, a normalization algorithm.

Remark. For convenience, our DRAMs (or DRAM_k) will sometimes use *multidimensional arrays.* It is not an essential extension of our machine model because a DRAM using a multidimensional array, for example, $A(i, j)$, can be simulated (in linear time) by a usual DRAM. Register $A(i, j)$ is represented by register $R(p)$, where $p = \text{Pair}(i, j)$ and Pair is an LTTC pairing function. That is (for example), $p = \text{Pair}(i, j) = i \mathbin{\hat{}} 12 \mathbin{\hat{}} \text{rep}(j)$ (concatenation), where $\text{rep}(j)$ is obtained by repeating twice each digit of j (see [Gr3] or [GrRo], [Ro1] for more details).

4.1. A normalization algorithm. In most of "linear time-bounded algorithms" on concrete data structures, such as graphs, formulas, and so on (cf. [AHU1], [AHU2], [FGS], [Kn],

[Me], [Se]), one implicitly assumes that input structures are initially normalized; for example, the set of vertices of a graph is the interval of integers $\{1, 2, \ldots, m\}$ and Dowling and Gallier [DoGa] and Minoux [Mi] assume that their satisfiability algorithm works on propositional Horn formulas whose variables are exactly $p_1, p_2, \ldots, p_m$.

However, it is not obvious how to transform an input, for example, a propositional formula into such a normalized one. Of course, we can list all the occurrences of variables p_i and then renumber them by sorting, but no linear time algorithm is known for the sorting problem. (Note that Itai and Makowsky [ItMa1] also mentioned the necessity of replacing large indices by small ones.)

We formalize the normalization problem as follows (assume we have fixed a finite alphabet $\sum = \{1, 2, \ldots, d\}$ that does not include symbols $\square$ and :).

PROBLEM NORMALIZE.

Input. A string S of the form $\square A_1 \square A_2 \ldots \square A_m$, where each $A_i \in \sum^*$. Let n denote the length of S ($n = m + \sum_{i \leq m}$ length (A_i)).

Output. A string S' of the form $\square A_1 : a_1 \square A_2 : a_2 \ldots \square A_m : a_m$, where

(i) for all i, j, $a_i = a_j$ if and only if $A_i = A_j$;

(ii) The a_i are integers such that $\{a_1, a_2, \ldots, a_m\} = \{1, 2, \ldots, M\}$

(a consequence of (i)–(ii) is $a_i = O(n/\log n)$ for each i);

(iii) length $(S') = O(n)$.

Remark. For each fixed alphabet $\sum$, the problem NORMALIZE is not unique. Our purpose is only to construct a DLINEAR algorithm that computes a string S' that satisfies (i–iii).

PROPOSITION 4.1. *Problem NORMALIZE belongs to DLINEAR.*

Proof. It is sufficient to prove that NORMALIZE belongs to $\mathrm{DRAM}_k(n/\log n)$ for sufficiently large k (we shall take $k \geq 2\log(d+1)$). Input S is initially divided into $b_k(n)$-blocks (in short, b-blocks with $b = b_k(n)$). Our DLINEAR algorithm will consist of two parts: first, lexical analysis of S and second, computation of the a_i (by hashing techniques).

Lexical analysis of S. In the sequel, a subword $\square A_i$ of S that includes at least the first symbol of a b-block in S will be called a "milestone." Lexical analysis consists in determining the milestones in S and dividing S as follows:

$$S = \square A_{h(1)} B_1 \square A_{h(2)} B_2 \ldots \square A_{h(q)} B_q,$$

where $\square A_{h(1)}, \square A_{h(2)}, \ldots, \square A_{h(q)}$ are the successive milestones (notice that $h(1) = 1$) and subwords $B_i \in (\sum \cup \{\square\})^*$ are the "remainders." For example, B_1 is the string of symbols (maybe empty) that lie between the first and second milestones A_1 and $A_{h(2)}$. We easily see that length$(B_i) < b$ (otherwise, B_i would contain the first symbol of a b-block and then it would include a milestone) and $q \leq$ (number of b-blocks) $= \ulcorner n/b \urcorner = O(n/\log n)$. It is easy to compute and store the list of milestones $\square A_{h(1)}, \ldots, \square A_{h(q)}$ (respectively, the list $B_1, B_2, \ldots, B_q$) in a two-dimensional (respectively, one-dimensional) array where each element has length $\leq b = \Theta(\log n)$. Notice that the total number of b-blocks in all the $A_{h(i)}$ is $O(n/b) = O(n/\log n)$.

Clearly a DRAM_k can perform the above computation in time $O(n/\log n)$. The lexical analysis of S is not complete because it does not separate the A_i contained in the B_j.

Computation of the a_i. We now use some techniques of [Gr3, paragraph III.3] (cf. [Wi2] for some comparable techniques). That requires some definitions: a subword A_i in S is "big" if length $(A_i) \geq b$; otherwise, it is "small." Note that each subword A_i of some B_j is small and that a word $A_{h(j)}$ (in a milestone $\square A_{h(j)}$) may be big or small. We do not separately encode each occurrence of a small A_i (it would require too many steps in case too many small A_i occur in S) but precompute them as follows.

In a first phase, recapitulate the small A_i that occur in S (use a boolean array, called PRESENT, indexed by words of $\sum^*$ of length less than b) and then encode them with an array denoted CODE (we take $a_i = \text{CODE}(A_i)$). The subroutine of this first phase is the following:

```
begin
        for each $w \in \sum^*$ such that length($w$) $< b$ do
                PRESENT($w$) := false {initialization}
        for $i := 1$ to $q$ do
                if length($A_{h(i)}$) $< b$, then PRESENT($A_{h(i)}$) := true
        for $i := 1$ to $q$ do
                if $B_i$ is "new" (i.e., $B_i \neq B_j$ for each $j < i$)
                then for each $A_j$ that occurs in $B_i$
                        do PRESENT($A_j$) := true
        {note: we check if $B_i$ is "new" by means of another boolean array indexed by
        words of length less than $b$}
        {we now encode "small" $A_i$ (that occur in $S$) with consecutive integers}
        $a := 1$
        for each $w \in \sum^*$ such that length($w$) $< b$ do
        if PRESENT($w$) then
                begin
                        CODE($w$) := $a${meaning: if $w = A_i$, then $a = \text{CODE}(A_i) = a_i$}
                        $a := a + 1$
                end
end;
```

Note that at each moment the global variable a represents the least integer that is not an encoding (it will be used in the third phase below).

In a second phase, encode all $B_i \in (\sum \cup \{\square\})^*$ in S as follows. If$B_i = \square\alpha_1\square\alpha_2 \ldots \square\alpha_s$, where each $\alpha_j \in \sum^*$, then take $\text{CODE}(B_i) := \square\alpha_1 : \text{CODE}(\alpha_1) \ldots \square\alpha_s : \text{CODE}(\alpha_s)$ (these encodings can be stored in the above CODE arrays, indexed by words of $(\sum \cup \{\square\})^*$ of length $< b$).

The reader should be convinced that first and second phases above require only $O(n/\log n)$ steps for sufficiently small $b = b_k(n)$, i.e., for sufficiently large k. In particular, the analysis of the A_j included in the "new" B_i requires time $O(b.(d+1)^b) = O((\log n)n^u)$, where $u = (1/k)\log(d+1)$ that is $O(n^{1/2}.\log n)$ if $k \geq 2\log(d+1)$. We can now compute the parts of the output S' that concern the subwords B_i and the milestones $\square A_{h(i)}$ for small $A_{h(i)}$. The third phase consists in encoding big milestones $\square A_j$ (i.e., for which length $(A_j) \geq b$) by techniques of [Gr3, paragraph III.3]. Let us describe it to have a self-contained proof.

Examine and encode successively the big words A_j by using two global variables denoted Nextp and Nexta and two "dynamic pairing functions" denoted PAIR and ENCODE (here we define a "pairing function" as an injective partial function $\mathbb{N} \times \mathbb{N} \to \mathbb{N} - \{0\}$). These functions are stored in two-dimensional arrays, also denoted PAIR and ENCODE, and each one is completed if necessary, i.e., when it is undefined on some useful arguments. At each moment, values of function PAIR (respectively, ENCODE) have to form an initial segment [1, Nextp - 1] (respectively, a segment [a, Nexta - 1], where a is the least positive integer that does not encode a little A_i; see first phase) and value Nextp (respectively, Nexta) is kept in a special register.

Let us explain how functions PAIR and ENCODE are computed and used. Let $u_1, u_2, \ldots, u_r$

denote the successive b-blocks of a big word A_j (length $(A_j) \geq b$) we want to encode. More precisely, $u = u_1 \hat{} \ u_2 \dots \hat{} \ u_r$ with $r \geq 1$, length$(u_i) = b$ for each $i < r$ and $0 <$ length$(u_r) \leq b$. Compute successively $p_2 = \text{PAIR}(p_1, u_2)$ where $p_1 = u_1$, $p_3 = \text{PAIR}(p_2, u_3), \dots, p_r = \text{PAIR}(p_{r-1}, u_r)$ and $e = \text{ENCODE}(p_r, r)$ in respecting the following rule: each time we need to use a pair of arguments (x, y) for which function PAIR is undefined (i.e., register PAIR(x, y) has never been visited and contains zero for that reason), execute the following assignments (and similarly for ENCODE):

$$\text{PAIR}(x, y) := \text{Nextp} \ ; \text{Nextp} := \text{Nextp} + 1.$$

At the beginning of the third phase, initialize variables by

$$\text{Nextp} := 1; \ \ \text{Nexta} := a.$$

Obviously, a nonzero integer is never contained in two distinct locations of the array PAIR (respectively, ENCODE); hence, PAIR and ENCODE are pairing functions. Let us define function PAIR^i : $\mathbb{N}^i \to \mathbb{N}$ by recurrence. PAIR$^i(x_1, x_2, \dots, x_i)$ is equal (if it is defined) to

$$\begin{aligned} &x_1 \quad \text{if } i = 1, \\ &\text{PAIR}(\text{PAIR}^{i-1}(x_1, \dots, x_{i-1}), x_i) \quad \text{if } i \geq 2. \end{aligned}$$

PAIR^i : $\mathbb{N}^i \to \mathbb{N}$ is clearly an injective partial function. In particular, we have considered (above) the following values associated with some big word $A_j = u_1 \hat{} \ u_2 \dots \hat{} \ u_r$; $p_2 = \text{PAIR}^2(u_1, u_2)$, $p_3 = \text{PAIR}^3(u_1, u_2, u_3), \dots p_r = \text{PAIR}^r(u_1, u_2, \dots, u_r)$ and $e =$ ENCODE (PAIR$^r(u_1, \dots, u_r), r)$.

We take $a_j := e$. Condition (i) of problem NORMALIZE is clearly respected. Notice that the last value of variable Nexta is $M + 1$, where M is the integer involved in Condition (ii).

We have now achieved the construction of output S'. One easily can be convinced that the third phase is performed within $O(n/\log n)$ steps (each b-block is manipulated $O(1)$ times) and only involves words of length $O(\log n)$. □

4.2. Other problems computable in linear time. In proofs that concrete problems belong to DLINEAR or NLINEAR we will often need the above NORMALIZE algorithm but also algorithms for the following problems.

PROBLEM LIST-COMPRESS.
Input. A string $S = \square A_1 \square A_2 \dots \square A_m$, where each $A_i \in \sum^*$ and $\square \notin \sum$.
Output. The string $S' = \square A'_1 \square A'_2 \dots \square A'_p$ such that:
(i) $\{A_1, A_2, \dots, A_m\} = \{A'_1, A'_2, \dots, A'_p\}$;
(ii) there is no repetition in S', i.e., $A'_i \neq A'_j$ if $i \neq j$;
(iii)A'_i occurs before A'_j in S' (i.e., $i < j$) iff the first occurrence of A'_i in S precedes the first occurrence of A'_j.

PROBLEM MULTISET-COMPRESS.
Input. A string $S = \square A_1 \square A_2 \dots \square A_m$, where each $A_i \in \sum^*$ and symbols □ and : do not belong to $\sum$.
Output. A string S' of the form $\square A'_1 : n_1 \square A'_2 : n_2 \dots \square A'_p : n_p$ such that
(i) $\{A_1, A_2, \dots, A_m\} = \{A'_1, A'_2, \dots, A'_p\}$;
(ii) $A'_1 \neq A'_j$ if $i \neq j$;
(iii) for each $i \leq p$, n_i is the number of indices j such that $A_j = A'_i$

PROBLEM ELEMENT-DISTINCTNESS.

Input. A string $\Box A_1 \Box A_2 \ldots \Box A_m$ where each $A_i \in \sum^*$ and $\Box \notin \sum$.

Question. Are the elements A_i all distinct?

Remark. The computational complexity of ELEMENT-DISTINCTNESS has been studied in some recent papers; for example, Yao [Ya] proves a near optimal time-space tradeoff for it.

PROBLEM PERMUTATION.

Input. A string $\Box A_1 \Box A_2 \ldots \Box A_m \Delta A'_1 \Box A'_2 \ldots \Box A'_m$, where A_i, A'_i are words of $\sum^*$ ($\Box$, $\Delta \notin \sum$).

Question. Is there a permutation π of $\{1, 2, \ldots, m\}$ such that $A'_i = A_{\pi(i)}$ for each $i \leq m$?

PROBLEM SET-INCLUSION (respectively, SET-EQUALITY).

Input. Two lists of words $A_1, A_2, \ldots, A_m$ and $A'_1, A'_2, \ldots, A'_p$.

Question. Does the inclusion $\{A_1, \ldots, A_m\} \subseteq \{A'_1, \ldots, A'_p\}$ (respectively, equality $\{A_1, \ldots, A_m\} = \{A'_1, \ldots, A'_p\}$) hold?

Remark. The input is presented above in a natural structured manner but formally the input of a DRAM is a string; the exact input of SET-INCLUSION has to be, for example, of the form $A_1 \Box A_2 \ldots \Box A_m \Delta A'_1 \Box A'_2 \ldots \Box A'_p$ (with new symbols $\Box$ and Δ). For convenience, in the sequel our problems will be presented in a structured manner.

PROBLEM CHECK-SORT.

Input. Two lists of integers (or words) $A_1, A_2, \ldots, A_m$ and $A'_1, A'_2, \ldots, A'_m$ such that $A'_1 \leq A'_2 \leq \cdots \leq A'_m$.

Question. Is the second list the sorted version of the first list, i.e., is there a permutation π of $\{1, 2, \ldots, m\}$ such that $A'_i = A_{\pi(i)}$ for each $i \leq m$?

PROBLEM PARTITION.

Input. A list P of "sets" $S_1, S_2, \ldots, S_m$, where each S_i is of the form: $S_i = \{e_1, e_2, \ldots, e_p\}$.

Question. Is P a "set partition," i.e., are the sets S_i mutually disjoint?

PROBLEM SET-INTERSECT (respectively, SET-DIFFERENCE).

Input. A set S and a list L of sets $S_1, S_2, \ldots, S_m$.

Output. The list L' of intersection sets $S_1 \cap S, \ldots, S_m \cap S$ (respectively, difference sets $S_1 - S, \ldots, S_m - S$).

PROBLEM ADDITION.

Input. A list of integers $A_1, A_2, \ldots, A_m$.

Output. The integer $A_1 + A_2 + \cdots + A_m$.

PROBLEM FUNCTION-DEF.

Input. A list of arrows $x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots, x_m \rightarrow y_m$, where the x_i, y_i are words.

Question. Does that list define a function, i.e., does the implication $x_i = x_j \Rightarrow y_i = y_j$ hold for all i, j?

PROBLEM FUNCTION-APPLY (Assume that c is a fixed integer).

Input. A "finite" partial function F on integers, i.e., a list of arrows $x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots, x_m \rightarrow y_m$, where words x_i are all distinct and length$(y_i) \leq c$ length(x_i) and a string of the form $\Box z_1 \Box z_2 \ldots \Box z_p$.

Output. The string $\Box z'_1 \Box z'_2 \ldots \Box z'_p$, where $z'_i = F(z_i)$ if $F(z_i)$ is defined (i.e., z_i is some x_j and $F(z_i) = y_j$) and $z'_i = z_i$ otherwise.

PROPOSITION 4.2. *Problems LIST-COMPRESS, MULTISET-COMPRESS, ELEMENT-DISTINCTNESS, PERMUTATION, SET-INCLUSION, and set-equality all belong to*

DLINEAR.

Proof. We only mention the differences with the proof of Proposition 4.1. After the lexical analysis of string S that produces (for instances of LIST-COMPRESS, MULTISET-COMPRESS, and ELEMENT-DISTINCTNESS) the decomposition

$$S = \Box A_{h(1)} B_1 \ldots \Box A_{h(q)} B_q$$

(as above), the output of LIST-COMPRESS is computed by the following loop:

```
For $i := 1$ to $q$ do
begin
    if $A_{h(i)}$is "new" {i.e., that is the first occurrence of $A_{h(i)}$ in $S$}
        then write $\Box A_{h(i)}$
    if $B_i$ is "new" {i.e., $B_i \neq B_j$ for each $j < i$}
        then for each $A_j$ that occurs in $B_i$ do
            if $A_j$ is "new" then write $\Box A_j$
end;
```

Note that we check if a small A_j is new with the boolean array PRESENT (of the proof of Proposition 4.1) indexed by words w, such that length$(w) < b$, and that a big A_j is new iff register ENCODE(PAIR$^r(u_1, \ldots, u_r), r)$ contains zero at that moment.

For MULTISET-COMPRESS, we compute the part of output S' concerning the big A_i by acceding to its encoding a_i (computed as usual) and to a counter COUNT(a_i) indexed by a_i. To produce the part of S' concerning the small A_i, we first compute for each word $w \in (\sum \cup \{\Box\})^*$ of length less than b, the number, denoted NUMBER(w), of indexes i such that $B_i = w$. Then for each word w of length less than b and of the form $\Box\alpha_1\Box\alpha_2 \ldots \Box\alpha_s$ with $\alpha_j \in \sum^*$, its contribution to the numbers of small A_i is counted by the loop

$$\text{for } j := 1 \text{ to } s \quad \text{do COUNTER}(\alpha_j) := \text{ COUNTER}(\alpha_j) + \text{ NUMBER}(w).$$

Finally, count the small $A_{h(i)}$ by the loop

```
for $i$ : = 1 to $q$ do
    if length $(A_{h(i)}) < b$  then COUNTER$(A_{h(i)})$ := COUNTER$(A_{h(i)}) + 1$.
```

This proves the proposition for MULTISET-COMPRESS.

Problem ELEMENT-DISTINCTNESS is obviously decided by MULTISET-COMPRESS algorithm. The PERMUTATION algorithm is similar to the previous one. The main difference is that the A_i and the A'_j are counted separately. At the end we check if the corresponding counters are equal. Problems SET-INCLUSION and SET-EQUALITY have similar algorithms. □

PROPOSITION 4.3. *Problem ADDITION belongs to DLINEAR.*

Proof. Without loss of generality, assume that the input is $\Box A_1\Box A_2 \ldots \Box A_m$. First, that input is transformed (as follows) in such a way that the number m of summands A_i becomes $O(n/\log n)$. After the lexical analysis that gives $\Box A_{h(1)}B_1\Box A_{h(2)}B_2 \ldots \Box A_{h(q)}B_q$ (cf. proof of Proposition 4.1), each string B_i is replaced by the sum of the integers that it contains; that is, more precisely, if $B_i = \Box\alpha_1\Box\alpha_2 \ldots \Box\alpha_s$, then B_i is replaced by string $\Box\ A'_i$, where A'_i is the result of the sum $\alpha_1 + \alpha_2 + \cdots + \alpha_s$. That can be done efficiently by computing previously for each w of the form $w = \Box\alpha_1\Box\alpha_2 \ldots \Box\alpha_s$ (where the α_i are integers), such that length$(w) < b$, the integer $\alpha_1 + \alpha_2 + \cdots + \alpha_s$, and storing it in a table SUM(w).

Second, add the $m = O(n/\log n)$ integers A_i by the usual classroom algorithm on b-blocks, that is, with integers in base d^b (recall that integers are originally written in d-adic notation). The reader should be easily convinced that it requires only $O(n/\log n)$ steps and involves integers of length $O(\log n)$ (because $b = \Theta(\log n)$). □

PROPOSITION 4.4. *Problems PARTITION, SET-INTERSECT, and SET-DIFFERENCE belong to DLINEAR.*

Proof. Let us consider PARTITION. Without loss of generality, assume that $m = O(n/\log n)$. (For that purpose, apply ELEMENT-DISTINCTNESS algorithm to the list P, where each S_i is regarded as a string, i.e., an ordered list. Reject if two lists S_i, S_j are equal.) A sufficiently large k is chosen to get a sufficiently small block length $b = b_k(n)$ (that will be precisely stated below).

Then use a deterministic Turing machine that separates the list P into two lists, P' and P'', such that $P' = (S'_1, S'_2, \ldots, S'_m)$, $P'' = (S''_1, S''_2, \ldots, S''_m)$, and each $S'_i = \{e \in S_i : \text{length}(e) \geq b\}$ and each $S''_i = \{e \in S_i : \text{length}(e) < b\}$.

Clearly, P is a partition if and only if both P' and P'' are partitions. The Turing machine runs in linear time $O(n)$ and can be simulated by a DRAM within time $O(n/\log n)$ (by Proposition 3.1 (iii)).

It is easy to check if P' is a partition by the above hashing techniques. Therefore, without loss of generality, assume that in $P = (S_1, S_2, \ldots, S_m)$, each S_i is a list of elements e such that length$(e) < b$. Then divide each string S_i into b-blocks, that is, into substrings $S_i = B_i^1 B_i^2 \ldots B_i^{p_i}$ (concatenation) where each block B_i^j is a sublist of S_i of the form $e_1, e_2, \ldots, e_s$ (each e_u is an "elementary element") such that length $(B_i^j) \leq b$. This can be done in such a way that the total number of blocks B_i^j (for all $i \leq m$ and $j \leq p_i$) is $O(n/\log n)$ (because $m = O(n/\log n)$).

Then for each block B_i^j, store (in an array denoted SET) the index $i = \text{SET}(B_i^j)$ of the set S_i where B_i^j is included; reject in case there is a collision, that is, $B_i^j = B_{i'}^{j'}$, for $i \neq i'$ (that means that S_i and $S_{i'}$ have common elements). Finally, for each possible pair of blocks B, B' that contain at least one common element e and such that $\text{SET}(B) \neq 0$ and $\text{SET}(B') \neq 0$ (i.e., both B and B' occur in P), check if $\text{SET}(B) = \text{SET}(B')$. That can be done in time $o(n/\log n)$ for sufficiently small block length b.

This proves that PARTITION belongs to DLINEAR. The algorithms and proofs for SET-INTERSECT and SET-DIFFERENCE are similar and then are left to the reader. □

PROPOSITION 4.5. *Problems CHECK-SORT, FUNCTION-DEF, and FUNCTION-APPLY are in DLINEAR.*

Proof. Problem CHECK-SORT is a restriction of the DLINEAR problem PERMUTATION.

Let us consider FUNCTION-DEF. Apply successively to its input the LIST-COMPRESS algorithm (to avoid repetitions of arrows) and the NORMALIZE algorithm. This gives a string of the form $a_1 \to b_1, a_2 \to b_2, \ldots, a_p \to b_p$, where $p = O(n/\log n)$ and each a_i, b_i is $O(n/\log n)$. Now by storing the arrows in an array F by assignments $F(a_i) := b_i$, check if there is no collision.

The construction of an algorithm for FUNCTION-APPLY is left to the reader. □

From the NORMALIZE algorithm, we deduce the following "thesis."

ASSERTION 4.6 (Linear Time Thesis). *"Concrete algorithms" usually regarded as "computable in linear time" and concerning (weighted, directed, nondirected) graphs given by their lists of edges or propositional formulas in clausal form are in DLINEAR.*

Remark. This statement is called an assertion or thesis because it seems to be vague. It can be *proved* only for specific problems, for example:

PROBLEM HORN-SAT [DoGa], [ItMa1], [ItMa2], [Mi].

Input. A set S of propositional Horn clauses $\{C_1, C_2, \ldots, C_m\}$.

Question. Is S satisfiable?

PROBLEM SCC ("strongly connected components" [Ta], [AHU1]).

Input. A directed graph $G = (V, E)$ given by its list of vertices $v_1, \dots, v_m$ and its list of arcs $e_1, \dots, e_p$ $(e_i \in V^2)$.

Output. The set of strongly connected components of graph G.

Proof of Assertion 4.6 *for SCC.* First normalize the input. Apply LIST-COMPRESS algorithm to remove repetitions in the list of vertices and the list of arcs; so, we can assume that $m + p = O(n/\log n)$. Apply NORMALIZE algorithm to these lists so that the set of vertices becomes $V = \{1, 2, \dots, m\}$. Then implement the usual algorithm of [Ta] (see also [AHU1], [AHU2]) on the DRAM model; it runs in time $O(m + p) = O(n/\log n)$. □

Proof of Assertion 4.6 *for HORN-SAT.* The algorithms of [DoGa], [ItMa1], [ItMa2], [Mi] can be trivially implemented on a DRAM in time $O(n/\log n)$ if in the list of clauses S

(i) each propositional variable has length $O(\log n)$, and

(ii) the total number of occurrences of variables is $O(n/\log n)$.

For that reason it is sufficient to exhibit a DRAM$(n/\log n)$ algorithm that computes from each input S a new set of Horn clauses S' for which (i)–(ii) hold and which is satisfiable if and only if S is satisfiable.

We can assume that each clause of S has one of the two following forms:

$$(v_1, \dots v_s \rightarrow v_0) : \text{ pure Horn clause;}$$
$$(v_1, \dots, v_s \rightarrow \text{ FALSE}) : \text{ negative clause,}$$

where each v_i is any propositional variable. The list of variables $v_1, \dots, v_s$ is called the *hypothesis list* of the clause. Let $C_1, C_2, \dots, C_m$ denote the list of clauses of S. By the LIST-COMPRESS algorithm, we can assume that there is no repeated clause so that $m = O(n/\log n)$.

Let us fix a sufficiently large integer k. We now separate in each hypothesis list the "big hypotheses," that is, variables v_i such that length$(v_i) \geq b = b_k(n)$, and the "small hypotheses," that is, the v_i such that length$(v_i) < b$. A linear time-bounded Turing machine can transform each clause into the form $(v_1, \dots, v_r, v_{r+1}, \dots, v_s \rightarrow u)$, where u is either a variable or the FALSE value and for $i \leq r$ (respectively, $i > r$), v_i is a big (respectively, small) hypothesis.

Then for each clause C_i $(i \leq m)$, divide its list of small hypotheses $L_i = v_{r+1}, \dots, v_s$ into "blocks": $L_i = B_i^1 B_i^2 \dots B_i^{q_i}$ (concatenation), where each "block" B_i^j is a sublist of L_i such that length$(B_i^j) \leq b$. This can be performed in such a way that the total number of sublists B_i^j (for $i \leq m$ and $j \leq q_i$) is $O(n/\log n)$ (because $m = O(n/\log n)$).

The next idea is to replace each B_i^j (in each clause C_i) by a new propositional variable, denoted v_i^j that will be interpreted as the conjunction of the variables of B_i^j. We store the new variable v_i^j in an array VARIABLE(B_i^j) so that if $B_i^j = B_{i'}^{j'}$, then v_i^j and $v_{i'}^{j'}$ denote the same variable. Let C_i' denote the clause C_i so transformed, that is,

$$C_i' = (v_1, \dots, v_r, v_i^1, v_i^2, \dots, v_i^{q_i} \rightarrow u).$$

Let S' be the set of clauses

$$S' = \{C_1', \dots, C_m'\} \cup \{B_i^j \rightarrow v_i^j\}_{i \leq m, j \leq q_i},$$

where repetitions among the clauses are removed. Note that there are as many distinct clauses $B_i^j \rightarrow v_i^j$ as distinct blocks B_i^j, that is, less then d^b (where d is the cardinality of the alphabet of clauses), which is $O(n^{1/2})$ for sufficiently small $b = b_k(n)$.

It remains to prove the two following claims.

Claim 1. The number of occurrences of propositional variables in S' is $O(n/\log n)$.

Claim 2. S is satisfiable if and only if S' is satisfiable.

Claim 1 follows from the following three facts:

- $m = O(n/\log n)$;
- there are $O(n/b)$ occurrences of hypotheses in $C'_1, \dots, C'_m$;
- there are $O(d^b) \cdot O(b) = O(n^{1/2} \cdot \log n)$ occurrences of variables in all the clauses of the form $B_i^j \rightarrow v_i^j$.

Claim 2 follows from the following two facts:

- S' implies S by transitivity of implication;
- conversely, if an interpretation satisfies S, then S' is trivially satisfied by the extended interpretation where each v_i^j is interpreted by the conjunction of the variables of B_i^j (i.e., v_i^j is true if and only if each variable of B_i^j is true). □

5. Linear time bounded reductions. In §§6 and 7, we will state that many NP-complete problems belong to NLINEAR or are complete in this class. In the proofs we use the following notions.

DEFINITIONS AND NOTATIONS. *Let $T(n) \geq n$ be a time function. A function on words (or a language) is in* DTIME($T(n)$) *(respectively,* NTIME($T(n)$)) *if it is computable (respectively, recognizable) by a deterministic (respectively, nondeterministic) Turing machine in time $O(T(n))$, where n is the length of the input word.*

Let A, $B \subseteq \sum^*$ be two languages. We write $B \in$ NTIME$^A(n)$ (respectively, $B \in$ NRAM$^A(n/\log n)$) or $B \leq_{\text{NTM}} A$ (respectively, $B \leq_{\text{NRAM}} A$) if there is a nondeterministic Turing machine (respectively, a NRAM*) M such that for each word $w \in \sum^n$: each computation of M on input w runs within time $O(n)$ (respectively, time $O(n/\log n)$) and w belongs to B iff some computation of M on w computes a word of A.*

In the special case when machine M is deterministic, we write $B \in$ DTIME$^A(n)$ (respectively, $B \in$ DRAM$^A(n/\log n)$) or $B \leq_{\text{DTM}} A$ (respectively, $B \leq_{\text{DRAM}} A$) and the function computed by M is called a reduction of B to A.

PROPOSITION 5.1. (i) DTIME(n) $\subseteq$ DRAM($n/\log n$); *in consequence, if $A \leq_{\text{DTM}} B$, then $A \leq_{\text{DRAM}} B$.*

(ii) *For each language A,* NTIME$^A(n) \subseteq$ NRAM$^A(n/\log n)$.

(iii) *If A, B are languages such that $A \in$ NTIME(n) (respectively, $A \in$ NRAM($n/\log n$)) and $B \in$ NTIME$^A(n)$ (respectively, $B \in$ NRAM$^A(n/\log n)$), then $B \in$ NTIME(n) (respectively, $B \in$ NRAM($n/\log n$)).*

(iv) *Relations $\leq_{\text{DTM}}$, $\leq_{\text{DRAM}}$, $\leq_{\text{NTM}}$, and $\leq_{\text{NRAM}}$ are transitive.*

(v) *Let $X \in \{D, N\}$ be a letter for "determinism" or "nondeterminism" and $T(n) \geq n$ (respectively, $T(n) \geq n/\log n$) be a nondecreasing function. If $A \leq_{\text{XTM}} B$ and $B \in$ XTIME($T(n)$) (respectively, $A \leq_{\text{XRAM}} B$ and $B \in$ XRAM($T(n)$)), then there is a constant c such that $A \in$ XTIME($T(cn)$) (respectively, $A \in$ XRAM($T(cn)$)).*

Proof. Assertions (i)–(v) either are classical results concerning Turing machines (see for example [HoUl], [Mo]) or are restatements or easy generalizations of assertions of Proposition 3.1. The proofs are similar. □

6. Most combinatorial NP problems belong to NLINEAR. We believe that the 21 NP-complete problems listed by Karp [Ka] are "combinatorial problems" (a possible exception is problem 19, called "job sequencing" in [Ka] and "sequencing to minimize tardy task weight" in [GaJo]; it seems to be more algebraic). The notion of combinatorial problem is as intuitive and at least as difficult to formulate as the notions of "computability" (captured by Turing computability) or "feasible computability" (captured by class P of polynomial time Turing computable problems). In the present section, we prove that many NP problems and, in particular, the 20 NP-complete problems 1–18 and 20 and 21 of [Ka] belong to NLINEAR. This justifies our conviction that combinatorial NP problems *are* NLINEAR problems. We choose to give explicit proofs for the following six significant problems.

SUBSET-SUM (Problem 18 in [Ka]).
Input. Integer B and list of integers $A_1, A_2, \ldots, A_m$.
Question. Is there a subset I of $\{1, 2, \ldots, m\}$ such that $\sum_{i \in I} A_i = B$?

EXACT-COVER (Problem 14 in [Ka]).
Input. List of sets $S_1, S_2, \ldots, S_m$ and set S.
Question. Is there a subset I of $\{1, 2, \ldots, m\}$ such that the family $(S_i)_{i \in I}$ is a partition of S?

HITTING-SET (Problem 15 in [Ka]; a variant numbered [SP8] appears in [GaJo]).
Input. List of sets $S_1, S_2, \ldots, S_m$.
Question. Is there a set S such that for each $i \leq m$, $S \cap S_i$ has exactly one element?

FEEDBACK-ARC-SET (Problem 8 in [Ka]).
Input. Integer K and directed graph $G = (V, A)$.
Question. Is there a subset $A' \subseteq A$ of cardinality at most K such that A' contains at least one arc from every directed cycle in G?

TRAVELING-SALESMAN (Problem [ND22] in [GaJo]).
Input. Integer K and weighted nondirected graph $G = (V, E)$ given by its list of weighted edges $e_1, e_1, \ldots, e_p$, where $e_i = (a_i, b_i, w_i)$, $a_i, b_i \in V$, and w_i is a positive integer.
Question. Has G a Hamilton circuit of weight not greater than K?

Pfleeger [Pf] has proved the NP completeness of the following problem.

PROBLEM RISA ([AL7] in [GaJo]).
Input. An integer K and an incompletely specified deterministic finite state automaton $\mathcal{A} = (Q, \sum, \delta, q_0, F)$, where Q is the set of states, $\sum$ is the input alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is a partial function: $Q \times \sum \rightarrow Q$.
Question. Can δ be extended to a total function $Q \times \sum \rightarrow Q$ in such a way that the minimal automaton of the resulting completely specified automaton has no more than K states?

Remark (RISA characterization). It is easy to see that the question above is equivalent to the following one (where Q' denotes the set of states reachable from q_0). Is there a map $h : Q' \rightarrow \{1, 2, \ldots, K\}$ that respects the following conditions?

(i) For all $q, q' \in Q'$ such that $h(q) = h(q')$, we have $q \in F$ if and only if $q' \in F$ (i.e., h respects the distinction "final/nonfinal states").

(ii) For all $q, q' \in Q', \sigma \in \sum$ such that $h(q)$ and $h(q')$ are equal and $\delta(q, \sigma)$ and $\delta(q', \sigma)$ are both defined, we have $h(\delta(q, \sigma)) = h(\delta(q', \sigma))$ (i.e., h respects the deterministic nature of the automaton).

THEOREM 6.1. *Twenty among the* 21 *NP-complete problems given by* [*Ka*] *that are problems* 1–18 *and* 20 *and* 21 *and problems TRAVELING-SALESMAN and RISA belong to NLINEAR.*

Remark. When an input involves a (weighted, directed, nondirected) graph, we assume that it is represented by its list of arcs.

Proof of Theorem 6.1. We prove the theorem for the six problems explicitly given above. The reader is invited to adapt the proofs to the other ones.

- TRAVELING-SALESMAN. Apply the FUNCTION-DEF algorithm (to check if no edge has two distinct weights) and the LIST-COMPRESS one (to suppress edge repetitions). By the NORMALIZE algorithm, we can assume that the set of vertices is $V = \{1, 2, \dots, m\}$ with $m = O(n/\log n)$. Then guess a cycle $e'_1, e'_2, \dots, e'_m$, where e'_i is a weighted edge: check if the cycle is included in the list of edges of G and includes each vertex (by two applications of SET-INCLUSION algorithm); check that no vertex is repeated in the cycle (by ELEMENT-DISTINCTNESS algorithm); and check if the weight of the cycle does not exceed K (by ADDITION algorithm).
- RISA. By successive applications of FUNCTION-DEF, LIST-COMPRESS, and NORMALIZE algorithms, normalize the input as above. In particular, we can assume that $\sum = \{1, 2, \dots, r\}$ and $Q = \{1, 2, \dots, s\}$ with $r + s = O(n/\log n)$ and that the number t of transitions of δ is $O(n/\log n)$. By a classical algorithm, a DRAM can compute the set Q' of reachable states within time $O(t) = O(n/\log n)$ and suppress the nonreachable states and the transitions that involve them. We now use the RISA characterization given in the remark above.

Guess for each reachable state $q \leq s$ a value $h(q) \leq K < s$ with which q is identified (function h can be stored in an array). Transform each transition $\delta(q, \sigma) = q'$ into transition $\delta'(h(q), \sigma) = h(q')$ and check if the resulting automaton (with set of states included in $\{1, 2, \dots, K\}$ and transition function δ') is deterministic (with FUNCTION-DEF algorithm) and has a well-defined set F' of final states: $h(q) \in F'$ if and only if $q \in F$.

- SUBSET-SUM. A linear time-bounded nondeterministic Turing machine (simulated by a NRAM in time $O(n/\log n)$) can guess a sublist $(A_i)_{i \in I}$. Then check if $\sum_{i \in I} A_i = B$ (by ADDITION algorithm).
- EXACT-COVER. Guess the sublist $(S_i)_{i \in I}$ as above and then check if it is a partition (by PARTITION algorithm) and if S is equal to the union set $\cup_{i \in I} S_i$ (by SET-EQUALITY algorithm).
- HITTING-SET. Guess a set S (of course, length$(S) \leq n$) and then check if for each $i \leq m$, set $S_i \cap S$ has exactly one element (by SET-INTERSECT algorithm).
- FEEDBACK-ARC-SET. The question is equivalent to the following one. Is there a subset $A' \subseteq A$ of cardinality at most K such that the directed graph $G' = (V, A - A')$ is acyclic? The recognizability of acyclicity of directed graphs belongs to DLINEAR (use some classical algorithm). □

The reader is invited to prove similarly that many other combinatorial NP problems (subgraph isomorphism, graph isomorphism, partition into cliques, and so on) are in NLINEAR.

7. Some NP-complete problems are NLINEAR complete. As seen in §6, the class NLINEAR = NRAM$(n/\log n)$ that refines the NP class includes many natural NP-complete problems. The following notion is comparable with NP completeness.

DEFINITION. *A problem A is NLINEAR complete if*

(i) *$A \in$ NLINEAR and*

(ii) *A is NLINEAR hard, that is, for each problem B in NLINEAR, $B \leq_{\mathrm{DTM}} A$.*

Remark. It is obvious that a NLINEAR-complete problem is also NP complete. Recall that $B \leq_{\mathrm{DTM}} A$ implies $B \leq_{\mathrm{DRAM}} A$. The following proposition is an immediate consequence of the definition above.

PROPOSITION 7.1. *Let A be a NLINEAR-complete problem. The following assertions are equivalent:*

(i) *NLINEAR = DLINEAR;*

(ii) *$A \in$ DLINEAR.*

Remark. We do not know whether NLINEAR $\neq$ DLINEAR. However, Paul et al. [PPST] have proved a similar assertion for Turing machines, that is, NTIME$(n) \neq$ DTIME(n). It implies that no DTIME(n) algorithm exists for any NLINEAR-hard problem.

The following proposition means that any efficient algorithm for a NLINEAR-complete problem can be used to solve any other NLINEAR problem *without loss of efficiency.*

PROPOSITION 7.2. *Let* $T(n) \geq n$ (*respectively,* $T(n) \geq n/\log n$) *be any nondecreasing function. If any NLINEAR-complete* (*or NLINEAR-hard*) *problem belongs to* DTIME($T(n)$) (*respectively,* DRAM($T(n)$)), *then NLINEAR* $\subseteq \cup_c$ DTIME($T(cn)$) (*respectively, NLINEAR* $\subseteq \cup_c$ DRAM($T(cn)$)).

Proof. Immediate. □

COROLLARY 7.3. *Let* A, B *be two NLINEAR-complete problems and* $T(n) \geq n$ *and* $T'(n) \geq n/\log n$ *be two nondecreasing functions. Then*

(i) $A \in \cup_c$ DTIME($T(cn)$) *if and only if* $B \in \cup_c$ DTIME($T(cn)$);

(ii) $A \in \cup_c$ DRAM($T'(cn)$) *if and only if* $B \in \cup_c$ DRAM($T'(cn)$).

It is well known that the problem SAT of Cook ("satisfiability of propositional formulas in clausal form") is a "generic" NP-complete problem [Co1], [Co2], [HoUl]. Similarly, we are going to use the following problem, denoted CONTRACT ("contraction of partial functional structures"), as a tool to prove NLINEAR completeness of other problems.

PROBLEM CONTRACT [Ra1], [Ra2].

Input. Set C of "constants," set X of "variables," set $\mathcal{F}$ of unary function symbols (elements of sets C, X, $\mathcal{F}$ are assumed to be presented in *lexicographical order*) and conjunction Γ of equalities of the form $f(u) = v$, where $f \in \mathcal{F}$, $u, v \in C \cup X$ and such that terms $f(u)$ are all distinct in Γ and occur in lexicographical order.

Question. Is conjunction Γ satisfiable on set C, that is, is there a function $VAL\colon C \cup X \to C$ that is the identity on C and, for each $f \in \mathcal{F}$, an interpretation f', that is, a partial function $f' : C \to C$ such that if $f(u) = v$ is an equality in Γ, then $f'(VAL(u)) = VAL(v)$ holds?

The following logical notion (a variant of a concept introduced by Scholz [Sz] and generalized by Fagin [Fa]) will be an essential tool in our proof that CONTRACT is a NLINEAR-complete problem.

DEFINITION. *Let* φ *be a first-order sentence* (*with equality*) *with type* $\mathcal{T} = \{f_1, f_2, \ldots, f_p, g_1, g_2, \ldots, g_q\}$ *where the* f_i, g_j *are unary function symbols, respectively, called specified and unspecified function symbols. The generalized spectrum of* φ, *denoted* GenSPECTRUM(φ), *is the set of structures* $(m, f_1, f_2, \ldots, f_p)$ (*where m is a positive integer identified with set* $\{0, 1, \ldots, m-1\}$) *that have an expanded structure* $(m, f_1, f_2, \ldots, f_p, g_1, g_2, \ldots, g_q)$ *that satisfies* φ.

Remark. It is convenient that an interpretation of a function symbol f_i, g_j will be denoted f_i, g_j (i.e., without any change).

Fagin, in his well-known seminal paper [Fa], proved that the class of generalized first-order spectra (in other words, the existential second-order logic) exactly captures the class NP. The following proposition refines Fagin's result in some manner: it shows that the subclass NLINEAR can be similarly captured by the existential second-order logic with only one (first-order) variable.

PROPOSITION 7.4. *If* $A \in$ *NLINEAR, then there is a first-order sentence* φ *of unary type* $\mathcal{T}$ *as above, such that*

(i) $A \leq_{\mathrm{DTM}}$ GenSPECTRUM(φ);

(ii) φ *is of the form* $\forall x \Psi(x)$, *where* $\Psi(x)$ *has only one variable and is a conjunction* $\bigwedge_i \sigma_i(x) = \tau_i(x)$ *of equalities of the form*

$$G_r \ldots G_2 G_1(x) = H_s \ldots H_2 H_1(x),$$

where $r \geq 1$, $s \geq 0$ *and each* G_i, $H_j \in \mathcal{T}$, *and such that* (ii.1) *no term or subterm of the form* $f_i(x)$ (*where* f_i *is a specified function symbol*) *occurs in* Ψ; *and*

(ii.2) *no first member* $\sigma_i(x)$ *is a subterm of another first member* $\sigma_j(x)$ (*for* $i \neq j$) *or of a second member* $\tau_j(x)$ (*here "subterm" means "proper subterm or equal term"*).

Sketch of proof. The proof is rather long and technical. It is essentially given in [Gr2], [Ra1], [Ra2] where the result is stated in a slightly weaker form. In those papers Language A is assumed to be in NTIME(n) (because the authors focus on Turing machines), but as a matter of fact, they prove the stronger result $A \leq_{\text{DTM}}$ GenSPECTRUM(φ) for all $A \in \text{NRAM}_k(n/\log n)$ (for sufficiently large k) and use the inclusion NTIME(n) $\subseteq \text{NRAM}_k(n/\log n)$. Let us roughly recall the stages of the proof (for details, the reader is invited to read the complete proof in [Gr2], [Ra1], [Ra2]).

First, construct a prenex first-order sentence φ that describes the computation of the NRAM$_k$ (which recognizes A in time $O(n/\log n)$). φ has to be interpreted on domain $m = \{0, 1, \dots, m-1\}$ where $0, 1, \dots, m-1$ intuitively represent the m successive instants of the computation (in particular, $m = O(n/\log n)$). Moreover, the prenex sentence φ has only one universally quantified variable and its only nonlogical symbols are $=$, $<$ (equality and natural order on m) and unary function symbols.

Second, transform formula φ in such a way that it satisfies condition (ii) above. In particular, natural order $<$, negations, and disjunctions are removed by some techniques comparable with Skolemization. New specified or unspecified function symbols are introduced for that purpose. □

Remark. We can assume that an output of the reduction $A \leq_{\text{DTM}}$ GenSPECTRUM (in Proposition 7.4), which is a structure $\mathcal{S} = (m, f_1, f_2, \dots, f_p)$ with length($\mathcal{S}$) $= O(n)$, is presented in the following manner: each function f_j is given in increasing order of its arguments, that is, by a list of the form $0 \to f_j(0)$, $1 \to f_j(1), \dots, m-1 \to f_j(m-1)$ (recall that $m = O(n/\log n)$ and the number p of functions does not depend on input w).

LEMMA 7.5. *Let* φ *be the sentence* $\forall x \Psi(x)$ *of Proposition* 7.4. *We have*

$$\text{GenSPECTRUM}(\varphi) \leq_{\text{DTM}} \textit{CONTRACT},$$

where each input structure $\mathcal{S} = (m, f_1, \dots, f_p)$ *is assumed to be presented in increasing order* (*cf. remark above*).

Proof. The main idea of the DTIME(n) reduction is the following one: unroll the sentence $\varphi = \forall x \Psi(x)$ on its domain m; that is, replace φ by conjunction $\bigwedge_{e<m} \Psi(e)$ where $\Psi(e)$ is obtained by substituting constant e for each occurrence of x in Ψ. From the definition of generalized spectrum, it is obvious that a structure $\mathcal{S}$ (on domain m) belongs to GenSPECTRUM(φ) if and only if the following conjunction of equalities, denoted γ, is satisfiable on domain m:

$$\bigwedge_{e<m} \Psi(e) \wedge \bigwedge_{j \leq p} \text{Diagram } (f_j)$$

where Diagram(f_j) denotes the conjunction of equalities $\bigwedge_{e<m} f_j(e) = e_j$ (sorted by increasing order of argument e) and e_j denotes the value of the term $f_j(e)$ in $\mathcal{S}$.

The conjunction γ is almost an instance of problem CONTRACT (with constants $0, 1, \dots, m-1$).We now have to "cut" the compositions of functions in each conjunct $\Psi(e)$ ($\Psi(e)$ is of the form $\bigwedge_i \sigma_i(e) = \tau_i(e)$). For that purpose introduce for each (sub)term $\mathcal{O}(e)$ in γ (except for constants $e < m$) a "variable" denoted $\underline{\mathcal{O}(e)}$ (underlined). The instance of problem CONTRACT corresponding to $\mathcal{S}$ will be the following one:

- $C = m = \{0, 1, \dots, m-1\}$;
- $X = \{\underline{\mathcal{O}(e)} : \mathcal{O}(e)$ is a (sub)term in γ, distinct from $e\}$;
- $\mathcal{F} = \{f_1, \dots, f_p, g_1, \dots, g_q\}$;

• conjunction Γ is obtained by unrolling each (sub)term in γ; for example, if γ contains the equality $f_1 g_3(2) = g_1(2)$, we replace that equality by the following conjunction: $g_1(2) = \underline{g_1(2)} \bigwedge g_3(2) = \underline{g_3(2)} \bigwedge f_1(\underline{g_3(2)}) = \underline{g_1(2)}$.

We easily see that conjunction γ is satisfiable on m iff Γ is satisfiable on m. Hence, the correspondence $\mathcal{S} \mapsto (C, X, \mathcal{F}, \Gamma)$ is a reduction of GenSPECTRUM(φ) to problem CONTRACT. That reduction is computable in linear time by a deterministic Turing machine. □

COROLLARY 7.6. *Problem CONTRACT is NLINEAR hard.*

PROPOSITION 7.7. *CONTRACT* $\leq_{\mathrm{DTM}}$*RISA.*

Proof. An instance $\mathbb{C} = (C, X, \mathcal{F}, \Gamma)$ of problem CONTRACT (without loss of generality, we assume $C = \{0, 1, \ldots, m-1\}$) is reduced to the following instance $(\mathcal{A}, K)$ of problem RISA:

$$\mathcal{A} = (Q, \textstyle\sum, \delta, q_0, F)$$

where

• $Q = C \cup \{m\} \cup X$;

• $\sum = \mathcal{F} \cup \{\mathrm{Succ}\}$ (intuitively, Succ is the successor function on integers);

• partial transition function $\delta : Q \times \sum \to Q$ is defined by $\delta(u, f) = v$ for each conjunct $f(u) = v$ in Γ (note that $\delta(u, f)$ is not defined twice because term $f(u)$ occurs at most once in Γ) and $\delta(u, \mathrm{Succ}) = u + 1$ for each $u \in C = \{0, 1, \ldots, m-1\}$ ($\delta(m, \mathrm{Succ})$ is undefined); $q_0 = 0$; $F = \{m\}$; and $K = m + 1$ (K is the cardinality of set $C \cup \{m\}$).

It is easy to see that each state $q \in Q$ is reachable from $q_0 = 0$ and that a deterministic Turing machine can compute the instance $(\mathcal{A}, K)$ in linear time $O(n)$. It remains to prove the following equivalence: $\mathbb{C} \in$ CONTRACT if and only if $(\mathcal{A}, K) \in$ RISA. From a solution h, for instance, $\mathbb{C}$ (i.e., a "contraction" of $\mathbb{C}$), we easily deduce a K-reduction of the incomplete automaton $\mathcal{A}$; that is, the contraction h of $\mathbb{C}$ on C that is extended by setting $h(m) = m$. The backward direction of the proposition, that is, $(\mathcal{A}, K) \in$ RISA implies $\mathbb{C} \in$ CONTRACT, is an easy consequence of the following lemma. This concludes the proof of Proposition 7.7. □

LEMMA 7.8. *Let $\mathcal{A}' = (Q, \sum, \delta', q_0, F)$ be a completely specified finite automaton that completes the above $\mathcal{A}$, that is, whose transition function δ' extends δ. In $\mathcal{A}'$*

(i) *all the states are reachable (from q_0);*

(ii) *the equivalence class of state m is $\{m\}$;*

(iii) *if $\mathcal{A}'$ is K-reducible (recall $K = m + 1$), then its K equivalence classes are exactly the class of* 0, *the class of* $1, \ldots,$ *the class of* $m - 1$, *and* $\{m\}$.

Proof. (i) holds for $\mathcal{A}$ and then for $\mathcal{A}'$. (ii) follows from the fact that m is the only final state. Let us prove (iii): we have $\delta(j, \mathrm{Succ}^{m-j}) = m \in F$ and $\delta(i, \mathrm{Succ}^{m-j}) = m - j + i \notin F$ for all $i < j \leq m$; hence, states i, j are not equivalent. □

COROLLARY 7.9. *Problems CONTRACT and RISA are NLINEAR complete.*

Proof. It follows from the fact that for each A in NLINEAR, $A \leq_{\mathrm{DTM}}$ CONTRACT $\leq_{\mathrm{DTM}}$ RISA and RISA $\in$ NLINEAR. □

Remark. Ranaivoson [Ra1], [Ra2] has strengthened Corollary 7.9. Problem RISA (respectively, CONTRACT) remains NLINEAR complete when it is restricted to a binary alphabet $\sum = \{0, 1\}$ (respectively, to type $\mathcal{T} = \{f_0, f_1\}$ with only two function symbols). He also shows that RISA is in P when $\sum = \{1\}$. Moreover, he exhibits several NLINEAR-complete problems concerning acyclic directed graphs. Note that [Ra1], [Ra2] shows the NLINEAR completeness of each above problem A by proving CONTRACT $\leq_{\mathrm{DTM}}$ A, but the reduction itself and the proof of its correctness are much longer and more technical than ours for

$A =$ RISA. Our student, Creignou [Cr], has recently proved the NLINEAR completeness of problems of directed or nondirected graphs by a simpler method.

Proposition 7.4 will be useful in the proof of the following characterization of class NLINEAR.

PROPOSITION 7.10. *NLINEAR = NTIME$^A(n)$ for all $A \in$ {FUNCTION-DEF, CHECK-SORT, PERMUTATION, SET-EQUALITY, SET-INCLUSION}.*

Proof. We have seen that each problem A above belongs to DLINEAR. Hence, NTIME$^A(n) \subseteq$ NLINEAR. To get the converse inclusion for $A =$ FUNCTION-DEF, it suffices to prove that for each first-order sentence $\varphi = \forall x \Psi(x)$ described in Proposition 7.4, we have GenSPECTRUM(φ) $\leq_{\mathrm{NTM}}$ FUNCTION-DEF. We show it by a variant of the proof of Lemma 7.5 (that states GenSPECTRUM(φ) $\leq_{\mathrm{DTM}}$ CONTRACT). Let us describe only the differences with that proof. Instead of unrolling the conjunction γ (associated with structure $\mathcal{S}$) by the introduction of "variables" for terms and subterms in γ, we now guess the intermediate values (less than m), that is, values of concerned (sub)terms. For example, if $m = 5$ and if γ contains the equality $f_1 g_3(2) = g_1(2)$, this equality may be replaced by the following conjunction

$$g_1(2) = \underline{4} \bigwedge g_3(2) = \underline{0} \bigwedge f_1(0) = 4$$

(convention: underlined $\underline{4}$ and $\underline{0}$ are "guessed"; the other occurrences 2, 0, and 4 are only "copied"). Let Γ denote the conjunction γ so transformed (there are as many possible Γ as possible guessed assignments). γ is satisfiable on m iff some associated Γ is satisfiable. Γ is almost an instance of FUNCTION-DEF, but it concerns not one but *several* functions $f_1, \ldots, f_p, g_1, \ldots, g_q$. For that reason, each equality $f_i(u) = v$ (respectively, $g_j(u) = v$) in Γ is replaced by arrow $u' \rightarrow v$, where u' is the integer $(p+q)u+i$ (respectively, $(p+q)u+p+j$) and u' encodes the term $f_i(u)$ (respectively, $g_j(u)$). Let Γ' denote the resulting instance of FUNCTION-DEF. Clearly, Γ is satisfiable if and only if $\Gamma' \in$ FUNCTION-DEF.

A nondeterministic Turing machine with input $\mathcal{S}$ can be constructed such that γ, Γ, and Γ' are successively computed within linear time. That proves GenSPECTRUM(φ) $\leq_{\mathrm{NTM}}$ FUNCTION-DEF.

To obtain the same results for any of the other four problems A, it suffices to show FUNCTION-DEF $\leq_{\mathrm{NTM}} A$. For any instance $L = (x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots, x_s \rightarrow y_s)$ of FUNCTION-DEF, guess some list $L = (x'_1 \rightarrow y'_1, \ldots, x'_s \rightarrow y'_s)$, where for each $i < s$: either $x'_i < x'_{i+1}$ or $x'_i = x'_{i+1}$ and $y'_i = y'_{i+1}$ (intuitively, L' defines a function and is sorted). Clearly, $L \in$ FUNCTION-DEF if there is some list L' such that $(L, L') \in$ CHECK-SORT (L' is the sorted version of list L). Moreover, a nondeterministic Turing machine can guess L' in linear time. Hence, FUNCTION-DEF $\leq_{\mathrm{NTM}}$ CHECK-SORT.

The same result is easily proved for problems PERMUTATION, SET-EQUALITY, and SET-INCLUSION with the same reduction. □

Remark. Proposition 7.10 holds for many other problems A. The abovementioned problems are the simplest ones we have found. In our opinion FUNCTION-DEF is the most significant. It indicates what computational power a nondeterministic linear time-bounded Turing machine lacks to capture NLINEAR.

COROLLARY 7.11. *NTIME(n) $\subseteq$ NLINEAR $\subseteq$ NTIME($n \log n$).*

Proof. The first inclusion has been proved above. The second one results from the following facts: NLINEAR $\leq_{\mathrm{NTM}}$ CHECK-SORT and CHECK-SORT $\in$ DTIME($n \log n$) (by the usual merge-sort algorithm). □

We have seen that several NP-complete problems (RISA, CONTRACT) are NLINEAR complete. The reader may ask whether most of natural NP-complete problems in NLINEAR are NLINEAR complete. We suggest a partial answer by studying problem SAT (i.e., satisfiability of propositional formulas in clausal form). We need the following mixed complexity class.

DEFINITION. *Let A be a language and $T(n)$, $T'(n)$ be time functions. We write*

$$A \in \mathrm{NDRAM}(T(n), T'(n))$$

if A is recognized by an NRAM *in such a way that for any input $w \in A$ of length n there is an accepting computation of the* NRAM *with only $O(T(n))$ GUESS instructions (i.e., nondeterministic steps) and $O(T'(n))$ other instructions (i.e., deterministic steps).*

PROPOSITION 7.12. *Problem SAT is in* $\mathrm{NDRAM}(n/(\log n)^2, n/\log n)$.

Proof. By running the NORMALIZE algorithm, we can assume that the set of propositional variables is $\{p_1, p_2, \ldots, p_r\}$, where $r = O(n/\log n)$. Guess a truth assignment $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_r) \in \{1, 2\}^r$ for the r variables (convention: $1 = \mathrm{TRUE}$, $2 = \mathrm{FALSE}$) by blocks of length $b = \ulcorner c \log(n+1) \urcorner$ (for a constant integer c). More precisely, a string $\alpha = B_1 B_2 \ldots B_s$, where each $B_i \in \{1, 2\}^*$, $\mathrm{length}(B_i) = b$ for $i < s$, and $0 < \mathrm{length}(B_s) \leq b$ ($s = \ulcorner r/b \urcorner$) is guessed and stored by s instructions $\mathrm{GUESS}(R(i))$ (B_i is stored in register $R(i)$).

The stored assignment α is analyzed by $O(r) = O(n/\log n)$ deterministic instructions that compute the list of pairs $(p_1, \alpha_1), (p_2, \alpha_2), \ldots, (p_r, \alpha_r)$.

Then run FUNCTION-APPLY algorithm to substitute for each variable p_i in each clause its truth value α_i. Finally, a finite automaton can evaluate all the clauses. Note that the number of nondeterministic instructions is $s = \ulcorner r/b \urcorner = O(n/(\log n)^2)$. □

COROLLARY 7.13. *The NLINEAR completeness of problem SAT would imply the following inclusion*: $\mathrm{NRAM}(n/\log n) \subseteq \mathrm{NDRAM}(n/(\log n)^2, n/\log n)$.

That inclusion is very unlikely: it would mean that any nondeterministic algorithm could be simulated by performing fewer nondeterministic steps and the same number (up to a multiplicative constant) of deterministic steps. We conjecture that that inclusion implies $P = NP$. (Indeed it seems stronger than $P = NP$.) Notice that Theorem 5 in [BuGo] presents a result (relative to an oracle) that, informally, shows that large amounts of nondeterminism can be useful even if small amounts are not.

Remark. Proposition 7.12 holds not only for SAT but also for many other NP-complete problems; in particular, it can be proved for five among the six problems studied in Theorem 6.1: TRAVELING-SALESMAN, SUBSET-SUM, EXACT-COVER, HITTING-SET, and FEEDBACK-ARC-SET (the proof is not trivial for TRAVELING-SALESMAN and SUBSET-SUM). The reader is encouraged to prove that and to discover other such problems.

Notice that Schnorr [Sr] and Cook [Co2] (respectively, Robson [Ro2]) have proved that any problem in $\mathrm{NTIME}(n)$ (respectively, computable by some nondeterministic RAM in linear time under logarithmic cost) is reducible to problem SAT in time $O(n(\log n)^2)$ on a deterministic Turing machine.

8. Conclusions and open problems. We have defined and studied two robust complexity classes: DLINEAR and NLINEAR. We believe that these classes satisfyingly capture the intuitive notions of deterministic and nondeterministic linear time.

- Each DLINEAR algorithm runs within linear time under logarithmic cost (hence, DLINEAR is included in the complexity class LINEAR defined in [Gr3]) and, similarly, for NLINEAR.
- DLINEAR contains most (all?) classical linear time-bounded algorithms (problems of graphs represented by their list of arcs, propositional Horn satisfiability, etc.) and several other ones (CHECK-SORT, LIST-COMPRESS, and so on).
- NLINEAR includes most of (all?) combinatorial NP-complete problems.

- The concepts we have introduced are natural and fruitful because there are *natural* NLINEAR-complete problems (the notion of NLINEAR completeness strengthens that of NP completeness) and then most combinatorial NP-complete problems are reducible to these problems in deterministic linear time.

The fact that many concrete algorithms run in linear time can be formalized in a uniform manner (i.e., independently of the nature of the inputs: graphs, lists, words, formulas) by using the DLINEAR concept. Similarly, we are convinced that our study of the class NLINEAR contributes to a better understanding of the theory of NP completeness of natural problems. In particular, it is interesting to notice that most of combinatorial NP-complete problems are in NLINEAR, which is a low (nondeterministic) complexity class.

We conclude this paper by describing some open problems and conjectures.

- Problem CHECK-SORT is in DLINEAR, but we have not been able to prove the same result for the sorting problem (our hashing techniques do not respect the natural order of integers).
- We strongly conjecture that DLINEAR includes the classical linear time-bounded algorithms when each graph is represented by its list of successors (respectively, predecessors), that is, by $(v_1, L_1), \ldots, (v_m, L_m)$ where L_i denotes the list of successors of vertex v_i. The difficult point is the fact that if such a representation has length n, then the number of distinct arcs of the graph is not necessarily $O(n/\log n)$ but may be $\Omega(n/\log\log n)$. We get over that difficulty by using the following remarks: it seems that each natural linear time algorithm on graphs is controlled either by depth-first search, by breadth-first search, or by topological sorting and it is very likely that the techniques of grouping "little vertices" into blocks can be naturally applied to these three searches in such a way that they work in time $O(n/\log n)$ on a DRAM.
- JOB-SEQUENCING is the only problem among the 21 NP-complete problems presented by Karp [Ka] for which we have not succeeded in proving that it belongs to NLINEAR. However, we strongly conjecture that it does (it seems to be more complicated and more algebraic than the 20 other ones. For example, partial sums occur in its statement).
- Class DLINEAR (respectively, NLINEAR) is contained in the class of problems computable in deterministic (respectively, nondeterministic) linear time under logarithmic cost. We conjecture that these classes are equal. Wiedermann [Wi2] has proved the following weaker result: a RAM that runs in time $T(n)$ under logarithmic cost can be simulated by a RAM that runs within time $O(T(n)/\log\log T(n))$ under uniform cost.
- We have proved that it can be required that NLINEAR (respectively, DLINEAR) computations involve only integers $O(n/\log n)$ (respectively, $O(n^{1+\epsilon})$); therefore, in the deterministic case, the DRAM may use somewhat "scattered" addresses $O(n^{1+\epsilon})$. It would be of great interest to show that the memory of a DRAM can be used in the most "compact" manner, that is, to require a DLINEAR computation to only involve addresses $O(n/\log n)$ like in the nondeterministic case.
- We have exhibited natural NLINEAR-complete problems. Similarly, the discovery of a natural *DLINEAR-complete* problem (i.e., a problem A such that DLINEAR $=$ DTIME$^A(n)$) would be a nice breakthrough; it would mean that *one* algorithm (the DLINEAR algorithm of A) would be the "pattern" of all the other DLINEAR algorithms (and they are numerous). We believe that such a result would follow from a logical characterization of class DLINEAR exactly as the NLINEAR completeness of problems CONTRACT and RISA has been deduced from our characterization of class NLINEAR by generalized spectra of first-order sentences.
- All the NLINEAR-complete problems we know concern contractions of structures (can a given structure be transformed by identification of some elements in such a way that the resulting structure respects some given constraint?) or satisfiability of formulas (cf. [Gr1]).

In fact, contraction problems can be regarded as satisfiability problems, and vice versa. Are there some NLINEAR-complete problems of some very different nature (e.g., subgraph isomorphism)?

- We conjecture that a careful analysis of the $\leq_{\mathrm{DTM}}$ reductions presented in §7 (to show that CONTRACT, RISA, and so on are NLINEAR hard) would prove that they are reset-loglin. This very strong reduction has been introduced by Compton and Henson [CoHe]. It strengthens the notion of logarithmic space linear time-bounded reduction. Moreover, the class of reset-loglin reductions is closed under composition.
- Does the NLINEAR completeness of problem SAT imply $P = NP$?

Acknowledgments. Thanks to Yuri Gurevich for stimulating discussions. Many thanks to Sylvie Hunout-Déroff for her help with the manuscript.

Note added in proof. Two abovementioned conjectures have been recently positively solved.

1. Grandjean (*Sorting, linear time, and the satisfiability problem*, to appear in a special issue of Annals of Math. and Artificial Intelligence) proved that the classical problem of SORTING *is* in DLINEAR and even belongs to the smaller class DLIN (DLIN is defined like DLINEAR but each RAM must only use integers $O(n/\log n)$ for any input of length n).

2. We recently noticed that JOB-SEQUENCING problem belongs to NLINEAR. This shows that *all* the NP-complete problems presented by Karp [Ka] *are* in NLINEAR.

REFERENCES

[AHU1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1974.

[AHU2] ———, *Data Structures and Algorithms*, Addison-Wesley, Reading, 1983.

[AnVa] D. Angluin and L. Valiant, *Fast probabilistic algorithms for Hamiltonian circuits and matchings*, J. Comput. System Sci., 18 (1979), pp. 155–193.

[BMS] A. Bertoni, G. Mauri, and N. Sabadini, *Simulations among classes of random access machines and equivalence among numbers succinctly represented*, Ann. Discrete Math., 25 (1985), pp. 65–90.

[BuGo] J. F. Buss and J. Goldsmith, *Nondeterminism within P*, STACS '91, Lecture Notes in Comput. Sci., 480 (1991), pp. 348–359.

[CoHe] K. J. Compton and C. W. Henson, *A uniform method for proving lower bounds on the computational complexity of logical theories*, Ann. Pure Appl. Logic, 48 (1990), pp. 1–79.

[Co1] S. A. Cook, *The complexity of theorem proving procedures*, Proc. 3rd Ann. ACM Symp. on Theory of Computing (1971), pp. 151–158.

[Co2] ———, *Short propositional formulas represent nondeterministic computations*, Inform. Process. Lett., 26 (1988), pp. 269–270.

[CoRe] S. A. Cook and R. A. Reckhow, *Time-bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[Cr] N. Creignou, *Temps linéaire et problèmes NP-complets*, Ph. D. thesis, Université de Caen, France, 1993.

[Cr2] ———, *Problems linearly equivalent to satisfiability or a uniform method to prove NP-completeness*, CSL'92, Lecture Notes in Comput. Sci., 702 (1993), pp. 115–133.

[De] A. K. Dewdney, *Linear time transformations between combinatorial problems*, Internat. J. Comput. Math., 11 (1982), pp. 91–110.

[DoGa] W. F. Dowling and J. H. Gallier, *Linear-time algorithms for testing the satisfiability of propositional Horn formulas*, J. Logic Programming, 3 (1984), pp. 267–284.

[Fa] R. Fagin, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computations, R. Karp, ed., American Mathematical Society, Providence, RI, SIAM-AMS Proc. Vol. VII, 1974, pp. 43–73.

[FGS] C. Froidevaux, M. C. Gaudel, and M. Soria, *Types de données et algorithmes*, McGraw-Hill, New York, Paris, 1990.

[GaJo] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, Freeman, San Francisco, 1979.

[Gl] E. Graedel, *On the notion of linear time computability*, Internat. J. Foundations Comput. Sci., 1 (1990), pp. 295–307.

[Gr1] E. GRANDJEAN, *A natural NP-complete problem with a nontrivial lower bound*, SIAM J. Comput., 17 (1988), pp. 786-809.

[Gr2] ———, *A nontrivial lower bound for an NP problem on automata*, SIAM J. Comput., 19 (1990), pp. 438–451.

[Gr3] ———, *Invariance properties of RAMs and linear time*, Comput. Complexity, to appear.

[GrRo] E. GRANDJEAN AND J. M. ROBSON, *RAM with compact memory: A realistic and robust model of computation*, CSL 90, Lecture Notes in Comput. Sci., 533 (1991), pp. 195–233.

[GuSh] Y. GUREVICH AND S. SHELAH, *Nearly linear time*, Lecture Notes in Comput. Sci., 363 (1989), pp. 108–118.

[HaSi] J. HARTMANIS AND J. SIMON, *On the power of multiplication in random access machines*, in Proc. 15th IEEE Symposium on Switching and Automata Theory, 1974, pp. 13–23.

[HPV] J. E. HOPCROFT, W. PAUL, AND L. VALIANT, *On time versus space and related problems*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.

[HoU1] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading, 1969.

[HuSt] H. B. HUNT AND R. E. STEARNS, *The complexity of very simple boolean formulas with applications*, SIAM J. Comput., 19 (1990), pp. 44–70.

[ItMa1] A. ITAI AND J. A. MAKOWSKY, *On the complexity of Herbrand's Theorem*, Computer Science Dpt, Technion, Haifa, Israel, Tech. rep. 243, May 1982.

[ItMa2] ———, *Unification as a complexity measure for logic programming*, J. Logic Programming, 4 (1987), pp. 105–117.

[Ka] R. M. KARP, *Reducibility among combinatorial problems, IBM Symp. 1972*, in Complexity of Computer Computations, Plenum Press, New York, 1972.

[KvLP] J. KATAJAINEN, J. VAN LEUWEN, AND M. PENTTONEN, *Fast simulation of Turing machines by random access machines*, SIAM J. Comput., 17 (1988), pp. 77–88.

[Kn] D. E. KNUTH, *The art of programming*, Vols. 1–3, Addison-Wesley, Reading, 1968, 1969, 1973.

[Me] K. MELHORN, *Data structures and algorithms*, Vol. 3, Springer-Verlag, New York, 1984.

[Mi] M. MINOUX, *LTUR: A simplified linear-time resolution algorithm for Horn formulae and computer implementation*, Inform. Process. Lett., 29 (1988), pp. 1–12.

[Mo] B. MONIEN, *About the derivation languages of grammars and machines*, 4th ICALP 1977, Lecture Notes in Comput. Sci., 52 (1977), pp. 337–351.

[PPST] W. PAUL, N. PIPPENGER, E. SZEMEREDI, AND W. T. TROTTER, *On determinism versus non-determinism and related problems*, in Proc. 24th IEEE Symp. Found. of Comput. Sci., 1983, pp. 429–438.

[Pf] C. P. PFLEEGER, *State reduction in incompletely specified finite-state machines*, IEEE Trans. Comput., 22 (1973), pp. 1099–1102.

[Ra1] S. RANAIVOSON, *Bornes inferieres non triviales powr des problemes naturels en theorie des graphes et des automates*, Ph. D. thesis, Université de Caen, France, 1990.

[Ra2] ———, *Nontrivial lower bounds for some NP-complete problems on directed graphs*, CSL 90, Lecture Notes in Comput. Sci., 533 (1991), pp. 318–339.

[Ro1] J. M. ROBSON, *Random access machines and multi-dimensional memories*, Inform. Process. Lett., 34 (1990), pp. 265–266.

[Ro2] ———, *An $O(T \log T)$ reduction from RAM computations to satisfiability*, Theoret. Comput. Sci., 82 (1991), pp. 141–149.

[Sr] C. P. SCHNORR, *Satisfiability is quasilinear complete in NQL*, J. Assoc. Comput. Mach., 25 (1978), pp. 136–145.

[Sc1] A. SCHOENHAGE, *Storage modifications machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

[Sc2] ———, *A nonlinear lower bound for random access machines under logarithmic cost*, J. Assoc. Comput. Mach., 35 (1988), pp. 748–754.

[Sz] H. SCHOLZ, *Ein ungelöstes Problem in der Symbolischen Logik*, J. Symbolic Logic, 17 (1952), p. 160.

[Se] R. SEDGEWICK, *Algorithms in C^{++}*, Addison-Wesley, Reading, MA, 1992.

[SlvEB] C. SLOT AND P. VAN EMDE BOAS, *The problem of space invariance for sequential machines*, Inform. and Comput., 77 (1988), pp. 93–122.

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

[WaWe] K. WAGNER AND G. WECHSUNG, *Computational complexity*, Reidel, Berlin, 1986.

[Wi1] J. WIEDERMANN, *Deterministic and nondeterministic simulation of the RAM by the Turing machine*, in Proc. IFIP Congress 83, R.E.A. Mason, ed., North Holland, Amsterdam, 1983, pp. 163–168.

[Wi2] ———, *Normalizing and accelerating RAM computations and the problem of reasonable space measures*, Lecture Notes in Comput. Sci., 443 (1990), pp. 125–138.

[Ya] A. C. YAO, *Near optimal time-space tradeoff for element distinctness*, in Proc. 29th IEEE Ann. Symp. Found. of Comput. Sci. 1988, pp. 91–97.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 598–616, June 1994

DIGITAL SEARCH TREES AGAIN REVISITED: THE INTERNAL PATH LENGTH PERSPECTIVE*

PETER KIRSCHENHOFER[†], HELMUT PRODINGER,[†] AND WOJCIECH SZPANKOWSKI[‡]

Abstract. This paper studies the asymptotics of the variance for the internal path length in a symmetric digital search tree under the Bernoulli model. This problem has been open until now. It is proved that the variance is asymptotically equal to $N \cdot 0.26600 + N \cdot \delta(\log_2 N)$, where N is the number of stored records and $\delta(x)$ is a periodic function of mean zero and a very small amplitude. This result completes a series of studies devoted to the asymptotic analysis of the variances of digital tree parameters in the symmetric case. In order to prove the previous result a number of nontrivial problems concerning analytic continuations and some others of a numerical nature had to be solved. In fact, some of these techniques are motivated by the methodology introduced in an influential paper by Flajolet and Sedgewick.

Key words. digital search trees, algorithm analysis

AMS subject classifications. 68Q25, 05C80

1. Introduction. Digital trees [2], [9], [17] are experiencing a new wave of interest due to a number of novel applications in computer science and telecommunications. For example, recent developments in the context of large external files and ideas derived from the dynamic hashing (virtual hashing, dynamic hashing, extendible hashing) led to the analysis of digital trees [8], [10], [12], [14], [15], [25], [29], [30], [31]. In telecommunications, recent developments in conflict resolution algorithms [22] and data compression [19] have also brought a new interest in digital trees. Some other applications are radix exchange sort, polynomial factorizations, simulation, Huffman's algorithm, and so on [2], [9], [17].

The three primary digital tree search methods are *digital search trees* (*DST*), *radix search tries* (shortly, tries), and *Patricia tries* [2], [7], [9], [17], [28], [31]. In the context of search costs one is led to investigate the *depth* of a node (search time) and the (external or internal) *path length* in digital trees. The average depth of a node for digital trees has been studied in [8], [17], [29], [30], [31], the variance in [12], [29], [30], [31], and limiting distributions in [10], [24], [25], [20]. The average value of the (external or internal) path length is closely related to the average depth of a node but *not* the variance. In [14], [15] the authors obtained the asymptotics of the variance for symmetric regular tries and Patricia tries, respectively (for asymmetric extensions of these results see [10]).

In this paper, we propose to evaluate the variance for the digital search trees, which has been an open problem until now. It has to be stressed that the variance of the internal path length in a digital search tree is the most difficult to estimate. This was already seen in the paper by Flajolet and Sedgewick [8], who establish an analytical methodology to analyze digital search trees (e.g., the average depth of a node). In this paper in the process of establishing the asymptotics of the internal path length we had to obtain some new analytic continuations of functions, which are mainly based on the famous Euler product identities. As in [12] and [13],

*Received by the editors June 25, 1990; accepted for publication (in revised form) March 10, 1993.

[†] Department of Algebra and Discrete Mathematics, Technical University of Vienna, Wiedner Hauptstrasse 8-10, A-1040 Vienna, Austria. This research was supported by Fonds zur Förderung der Wissenschaftlichen Forschung Projekt No. P7497-TEC.

[‡] Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. This author's research was supported by the National Science Foundation grants CCR-8900305 and its extension, CCR-9201078; International grant 8912631 and Networking and Communication Research grant 9206315; Air Force Office of Scientific Research grant 90-0107; North Atlantic Treaty Organization collaborative grant 0057/89; and National Library of Medicine grant R01 LM05118. This paper was revised while the author was visiting Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, France, and the research supported by Institute de Recherche d'Informatique et d'Automatique (projects ALGO, MEVAL, and REFLECS).

to derive the final results, namely, to show the cancellation of the higher order asymptotics, we had to appeal to the theory of modular functions (cf. §3). In addition, this problem possesses nontrivial numerical challenge. A very preliminary version of our results was presented at the 1989 IFIP Congress [16].

This paper is organized as follows. In the next section, we define our model, establish the general methodology to attack the problem, and present our main results. In particular, we show that the variance of the internal path length for the *binary symmetric* digital search tree under the Bernoulli model is asymptotic to $N \cdot 0.26600 + N \cdot \delta(\log_2 N)$, where N is the number of records and $\delta(x)$ is a periodic function with a very small amplitude. Section 3 contains proofs of our main results, followed by a few concluding remarks in §4.

2. Main results. Let $\mathcal{D}_N$ be the family of digital search trees built from N records with keys from a random stream of bits. Under the Bernoulli model, a key consists of 0's and 1's with independent and equal probability of appearance. Let L_N denote the random variable "internal path length"of trees in $\mathcal{D}_N$ and $F_N(z)$ the corresponding probability generating functions, i.e., the coefficient $[z^k]F_N(z)$ of z^k in $F_N(z)$ is the probability that a tree in $\mathcal{D}_N$ has internal path length equal to k. Then the following recursion, which is a direct consequence of the definition, holds:

$$F_{N+1} = z^N \sum_{k=0}^{N} 2^{-N} \binom{N}{k} F_k(z) F_{N-k}(z), \qquad F_0(z) = 1. \tag{2.1}$$

The *expectation* l_N is given by $l_N = F'_N(1)$ and fulfills for $N \geq 0$

$$l_{N+1} = N + 2^{1-N} \sum_{k=0}^{N} \binom{N}{k} l_k, \qquad l_0 = 0. \tag{2.2}$$

This recursion may be solved explicitly by using *exponential generating functions.* With $L(z) = \sum_{N \geq 0} l_N z^N / N!$, (2.2) translates into the functional differential equation

$$L'(z) = ze^z + 2e^{z/2} L(z/2).$$

By the substitution $\hat{L}(z) = e^z L(-z)$ we have the easier equation

$$\hat{L}(z) - \hat{L}'(z) = -z + 2\hat{L}(z/2).$$

With $\hat{L}(z) = \sum_{N \geq 0} \hat{l}_N z^N / N!$ we find for $N \geq 2$

$$\hat{l}_N = Q_{N-2}, \qquad \hat{l}_0 = \hat{l}_1 = 0$$

with the finite product

$$Q_N = \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{4}\right) \cdots \left(1 - \frac{1}{2^N}\right) \tag{2.3}$$

so that finally

$$l_N = \sum_{k=2}^{N} \binom{N}{k} (-1)^k Q_{k-2}. \tag{2.4}$$

The reader should note that an asymptotic evaluation of (2.4) is nonelementary due to the fact that terms of almost-equal magnitude occur with alternating signs. For this reason sophisticated methods from complex analysis are needed to find the correct order of growth. An essential step is the application of the following lemma from the calculus of finite differences.

LEMMA 1 (cf. [17, p. 38], [23]). *Let $\mathcal{C}$ be a path surrounding the points $j, j+1, \dots, N$ and $f(z)$ be analytic inside $\mathcal{C}$ and continuous on $\mathcal{C}$. Then*

$$\sum_{k\geq j}\binom{N}{k}(-1)^k f(k) = -\frac{1}{2\pi i}\int_{\mathcal{C}}[N;z]f(z)\,dz \tag{2.5}$$

with

$$[N;z] = \frac{(-1)^{N-1}N!}{z(z-1)\cdots(z-N)}.$$

In our application $f(z)$ is a meromorphic function that continues a sequence $f(k)$, e.g., $j=2$ and $f(k)=Q_{k-2}$ in (2.4). Moving the contour of integration, one can obtain the asymptotic expansion of the alternating sum by Cauchy's residue theorem, that is, for any real c (2.5) becomes $\sum_{k\geq j}\binom{N}{k}(-1)^k f(k) = \sum_{z_i\in\mathcal{P}_c}\operatorname{Res}\bigl([N;z_i]f(z_i)\bigr) + \mathcal{O}(N^c)$, where the sum is taken over the set of poles $\mathcal{P}_c$ different from $j, j+1, \dots, N$ with real part larger than c.

We note that the function $f(k) = Q_{k-2}$ possesses the analytic continuation $Q_z = Q_\infty/Q(2^{-z})$ where $Q(t) = \prod_{i\geq 1}(1-t/2^i)$ [8]. Then, applying a refinement of the technique of Flajolet and Sedgewick, we can easily prove the following theorem (cf. §3).

THEOREM 2. *The expectation l_N of the internal path length of digital search trees built from N records fulfills*

$$\begin{aligned} l_N &= N\log_2 N + N\left[\frac{\gamma-1}{\log 2} + \frac{1}{2} - \alpha + \delta_1(\log_2 N)\right] + \log_2 N \\ &\quad + \frac{2\gamma-1}{2\log 2} + \frac{5}{2} - \alpha + \delta_2(\log_2 N) + \mathcal{O}(\log N/N) \end{aligned} \tag{2.6}$$

with $\gamma = 0.57721\ldots$ (Euler's constant) and $\alpha = \sum_{n\geq 1} 1/(2^n-1) = 1.60669\ldots$, where $\delta_1(x)$ and $\delta_2(x)$ are continuous periodic functions of period 1, mean 0, and very small amplitude ($< 10^{-6}$). For later use we mention the Fourier expansion of $\delta_1(x)$

$$\delta_1(x) = \frac{1}{\log 2}\sum_{k\neq 0}\Gamma\left(-1-\frac{2k\pi i}{\log 2}\right)e^{2k\pi i x}, \tag{2.7}$$

where $\Gamma(x)$ is the gamma function [1].

We mention in passing that the $\mathcal{O}(1)$-term in (2.6) is slightly incorrect in [17].

Now we turn to the *analysis of the variance*, which is given by $\operatorname{Var} L_N = s_N + l_N - l_N^2$ with $s_N = F_N''(1)$. From (2.1) we get the recurrence relation (for $N\geq 0$; $s_0 = 0$)

$$\begin{aligned} s_{N+1} &= N2^{2-N}\sum_{k=0}^{N}\binom{N}{k}l_k + N(N-1) \\ &\quad + 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}l_k l_{N-k} + 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}s_k. \end{aligned} \tag{2.8}$$

In order to find an explicit solution to this recurrence, we split it into three parts: $s_N = u_N + v_N + w_N$, where

$$u_{N+1} = 2N(l_{N+1}-N) + 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}u_k, \quad N\geq 0,\ u_0 = 0, \tag{2.9a}$$

$$v_{N+1} = N(N-1) + 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}v_k, \quad N \geq 0,\ v_0 = 0, \tag{2.9b}$$

$$w_{N+1} = 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}l_k l_{N-k} + 2^{1-N}\sum_{k=0}^{N}\binom{N}{k}w_k, \quad N \geq 0,\ w_0 = 0. \tag{2.9c}$$

All of the above recurrences, as well as that for the average internal path length (2.2), fall into the following general recurrence studied in [30]. Let (x_n) be a sequence of numbers satisfying

$$x_{n+1} = a_{n+1} + 2^{1-n}\sum_{k=0}^{n}\binom{n}{k}x_k, \qquad n \geq 2, \tag{2.10}$$

where (a_n) is any sequence of numbers. The solution of (2.10) depends on the so-called *binomial inverse relations* that are defined as

$$\hat{a}_n = \sum_{k=0}^{n}(-1)^k\binom{n}{k}a_k \quad \text{and} \quad a_n = \sum_{k=0}^{n}(-1)^k\binom{n}{k}\hat{a}_k.$$

The second equation justifies the name binomial inverse relations. For more details, see Riordan [26]. A similar treatment as in the case of (2.2) leads to the following explicit solution (for details see [30]).

LEMMA 3. *Let* $x_0 = x_1 = 0$. *Then the recurrence* (2.10) *possesses the solution*

$$x_n = x_0 + n(x_1 - x_0) - \sum_{k=2}^{n}(-1)^k\binom{n}{k}\hat{x}_{k-2} \tag{2.11a}$$

where

$$\hat{x}_n = Q_n\sum_{i=1}^{n+1}\left[\hat{a}_i - \hat{a}_{i+1} - a_1\right]/Q_{i-1} \tag{2.11b}$$

and Q_n *is defined in* (2.3).

Using Lemma 3 we immediately solve our recurrences (2.9a)—(2.9c). In particular, one proves

$$\hat{u}_k = 2Q_{k-2}\left\{4 + \sum_{j=1}^{k-2}\frac{1}{2^j - 1} - \sum_{j=1}^{k-2}\frac{j}{2^j - 1} - \frac{2k}{2^{k-2} - 1}\right\}$$

$$\text{for } k \geq 3,\ \hat{u}_0 = \hat{u}_1 = \hat{u}_2 = 0; \tag{2.12a}$$

$$\hat{v}_k = -4Q_{k-2} \quad \text{for } k \geq 3,\ \hat{v}_0 = \hat{v}_1 = \hat{v}_2 = 0; \tag{2.12b}$$

and

$$\hat{w}_k = -Q_{k-2}\sum_{j=4}^{k-1}\frac{2^{1-j}}{Q_{j-1}}\sum_{i=2}^{j-2}\binom{j}{i}Q_{i-2}Q_{j-i-2}$$

$$\text{(2.12c)} \qquad \text{for } k \geq 5, \ \hat{w}_0 = \cdots = \hat{w}_4 = 0.$$

Of course, the "unhatted" solutions u_N, v_N, and w_N follow from the binomial relations, as shown in (2.11a). It is also worth mentioning that the recurrence for v_N is easy, and after simple algebra one proves

$$\text{(2.12d)} \qquad v_N = 4\binom{N}{2} - 4l_N,$$

so the treatment of u_N and w_N remains to be done.

In principle, u_N and w_N may be analyzed by making use of Lemmas 1 and 3. However, it turns out to be a highly nontrivial problem to find an analytical continuation of $\hat{w}_k$. After lengthy and difficult computations the residue calculus leads us to the following main result of this paper, which is proved in the next section.

THEOREM 4. *The variance of the internal path length of digital search trees built from N records becomes*

$$\operatorname{Var} L_N = N \cdot \{C + \delta(\log_2 N)\} + \mathcal{O}(\log^2 N/N)$$

where C is a constant that can be expressed as

$$\begin{aligned} C = {} & -\frac{28}{3L} - \frac{39}{4} - 2\beta_1 + \frac{2\alpha}{L} + \frac{\pi^2}{2L^2} + \frac{2}{L^2} - \frac{2}{L}\sum_{k\geq 3} \frac{(-1)^{k+1}(k-5)}{(k+1)k(k-1)(2^k-1)} \\ \text{(2.13)} \qquad & + \frac{2}{L}\sum_{r\geq 1} b_{r+1}\left(\frac{L(1-2^{-r+1})/2-1}{1-2^{-r}} - \sum_{k\geq 2}\frac{(-1)^{k+1}}{k(k-1)(2^{r+k}-1)}\right) \\ & + \frac{2}{L}\hat{w}'(3) - 2[\delta_1\delta_2]_0 - [\delta_1^2]_0 \end{aligned}$$

with $L = \log 2$, $\alpha = \sum_{n\geq 1} 1/(2^n - 1)$, $\beta_1 = \sum_{n\geq 1} n2^n/(2^n-1)^2$, *and* $b_{r+1} = (-1)^r 2^{-\binom{r+1}{2}}$. *The fluctuating function* $\delta(x)$ *is continuous with period* 1, *mean zero, and* $|\delta(x)| \leq 10^{-6}$, *and* $|[\delta_1^2]_0| \leq 10^{-10}$, *and* $|[\delta_1\delta_2]_0| \leq 10^{-10}$. *Finally,* $\hat{w}(z)$ *is a function defined as*

$$\text{(2.14)} \quad \frac{\hat{w}(z+1)}{Q_{z-1}} = -2Q_\infty z + \frac{\xi(z+2)}{2^z Q_z} + \frac{\xi(z+3)}{2^{z+1}Q_{z+1}} + \sum_{j\geq 2}\left(\frac{\xi(z+j+2)}{2^{z+j}Q_{z+j}} - \frac{\xi(j+2)}{2^j Q_j}\right)$$

with $Q_z = Q_\infty/Q(2^{-z})$, *where* $Q(t) = \prod_{i\geq 1}(1 - t/2^i)$, $Q_\infty = Q(1)$, *and*

$$\begin{aligned} & \xi(z+1) = \sum_{r\geq 0}\frac{b_{r+1}}{Q_r}\cdot\frac{Q_\infty}{Q(2^{3-z-r})} \\ \text{(2.15)} \qquad & \cdot\left\{2^z - \frac{2}{1-2^{1-z-r}} - \frac{2z}{1-2^{2-z-r}} + 2\sum_{k\geq 2}\binom{z}{k}\frac{1}{2^{r+k-1}-1}\right\}. \end{aligned}$$

Numerical evaluation of the constant C reveals that $C = 0.26600\ldots$ *and all five digits after the decimal point are significant.*

We should point out that in order to achieve the same accuracy in C one needs to run the recurrence equations (2.9a)–(2.9c) for $N \sim 10^6$.

In the following lemma we present an explicit formula for $\hat{w}'(3)$ that is convenient for numerical evaluations.

LEMMA 5. *The following identity holds*:

$$\begin{aligned}\frac{2\hat{w}'(3)}{L} = &-\frac{2Q_\infty}{L} + \sum_{j\ge2}\frac{1}{2^jQ_j}\sum_{r\ge0}a_{r+1}Q_{r+j-2}\\ &\cdot\Bigg\{-\sum_{n\ge1}\frac{1}{2^{n+r+j+2}-1}\cdot\left(2^{j+1}-2j-4+2\sum_{k=2}^{j-1}\binom{j+1}{k}\frac{1}{2^{r+k-1}-1}\right)\\ &+\frac{2}{(1-2^{-j-r})^2}+\frac{2j+2}{(1-2^{1-j-r})^2}-\frac{2}{L}\frac{1}{1-2^{1-j-r}}\\ &-2\sum_{k=2}^{j+1}\binom{j+1}{k}\frac{1}{2^{r+k-1}-1}+\frac{2}{L}\sum_{k=1}^{j+1}\binom{j+1}{k}\frac{1}{2^{r+k}-1}\\ &+\frac{2}{L}\sum_{k=0}^{j+1}\binom{j+1}{k}\sum_{i\ge1}\frac{(-1)^i}{(i+1)(2^{r+k+i}-1)}\Bigg\}\\ &+\sum_{j\ge3}\frac{\xi(j+2)}{2^jQ_j}\sum_{k\ge j+1}\frac{1}{2^k-1}\end{aligned} \tag{2.16}$$

where

$$a_{r+1} = \frac{b_{r+1}}{Q_r} = (-1)^r 2^{-\binom{r+1}{2}}/Q_r \tag{2.17}$$

and

$$\xi(j+2) = \sum_{k=2}^{j-1}\binom{j+1}{k}Q_{k-2}Q_{j-k-1} \tag{2.18}$$

with Q_k defined in Theorem 4.

Before we proceed to the proof of our results, we first offer some remarks and extensions.

Theorem 4 and our previous results in [12] and [30] provide asymptotics for the average covariance between two different nodes in a digital search tree (DST). Let $D_N^{(i)}$ be the length of the path from the root to the ith node, i.e., the node corresponding to the ith key of the N keys. Observe that the distribution of the random variable defined by $D_N^{(i)}$ depends on i for DST, since, in contrast to tries and Patricia tries, the order of insertion of the keys is relevant in this instance. Note that the internal path length L_N is expressed in terms of $D_N^{(i)}$ as

$$L_N = \sum_{i=1}^{N} D_N^{(i)}. \tag{2.19}$$

In [12] it has been shown that, besides a small fluctuation,

$$A_N = \frac{1}{N}\sum_{i=1}^{N}E\left(D_N^{(i)2}\right) - \left(\frac{1}{N}\sum_{i=1}^{N}ED_N^{(i)}\right)^2 \sim 2.844\dots\,. \tag{2.20}$$

(The quantity NA_N was erroneously identified with the variance of the path length.) It is easily checked that

$$\sum_{i=1}^{N}\text{Var }D_N^{(i)} - NA_N = \frac{1}{N}\left(\sum_{i=1}^{N}ED_N^{(i)}\right)^2 - \sum_{i=1}^{N}\left(ED_N^{(i)}\right)^2. \tag{2.21}$$

The expectations $ED_N^{(i)} = ED_i^{(i)}$ (for $N \geq i$) may now be computed in a straightforward manner. Indeed,

$$ED_N^{(N)} = E\left(L_N - L_{N-1}\right) = -\sum_{k=1}^{N-1}\binom{N-1}{k}(-1)^k Q_{k-1}, \tag{2.22}$$

and we find

$$ED_N^{(N)} = \log_2 N + \frac{\gamma}{L} + \frac{1}{2} - \alpha + \tau(\log_2 N) + O\left(\frac{1}{N}\right) \tag{2.23}$$

with a small fluctuation $\tau(x)$. Altogether we get, besides the fluctuating term,

$$\sum_{i=1}^{N} \operatorname{Var} D_N^{(i)} - NA_N \sim -\frac{N}{L^2}, \tag{2.24}$$

and therefore, the average variance of the depth becomes

$$\frac{1}{N}\sum_{i=1}^{N} \operatorname{Var} D_N^{(i)} \sim 0.763\dots . \tag{2.25}$$

(Compare also [27].) Now

$$\operatorname{Var} L_N = \sum_{i=1}^{N} \operatorname{Var} D_N^{(i)} + 2\sum_{i \neq j} \operatorname{Cov}\left\{D_N^{(i)}, D_N^{(j)}\right\}.$$

Using Theorem 4 and the above we find that the average value of

$$\operatorname{Cov}\{D_N^{(i)}, D_N^{(j)}\} \text{ is } \sim -0.50\dots/N.$$

The last statement says that, on the average, $D_N^{(i)}$ and $D_N^{(j)}$ are negatively correlated. Note that the equivalent quantity for regular tries is approximately equal to $+0.84\dots/N$ [14] and for Patricia $= -0.63\dots/N$ [15].

In order to select the best digital tree one needs to compare different characteristics of digital trees, namely regular tries, Patricia tries, and Digital Search Trees (DST). Table 1 contains four important parameters that are often used to predict a random shape of these trees (cf. [8], [12], [14], [15], [17], [18], [29], [30], [31]).

TABLE 1

	$\mathrm{E}L_n$	$\mathrm{Var}L_N$	"average" $\mathrm{Var}D_N^{(i)}$	"average" $\mathrm{Cov}(D_N^{(i)}, D_N^{(j)})$
DST	$N(\log_2 N - 1.71)$	$N \cdot 0.26$	0.763	$-0.50/N$
TRIES	$N(\log_2 N + 1.33)$	$N \cdot 4.35$	3.507	$+0.84/N$
PATRICIA	$N(\log_2 N + 0.33)$	$N \cdot 0.37$	1.000	$-0.63/N$

It can be seen from the table that the average external (internal) path length is approximately the same for all three digital trees. However, the variance of the depths and internal (external) path lengths differ significantly. We also notice that the variance of the internal path length for DST as well as the variance of the depth are smaller than the respective quantities for Patricia.

We point out that from Theorem 4 it follows that L_N/EL_N tends to one *almost surely* (i.e., with probability one) as $N \to \infty$. This immediately follows from Theorem 4 and the Borel-Cantelli lemma. (Compare, e.g., [21] for this standard argument.)

Finally, Theorem 4 provides the missing coefficient in the variance of the numbers of phrases in the Lempel-Ziv parsing algorithm [3], [19] (cf. also [11]). More precisely, the Lempel-Ziv parsing algorithm partitions a single string into variable phrases (blocks) such that a new block is the shortest substring not seen in the past as a phrase. For example, the string 11001010001000100 0 is parsed into (1)(10)(0)(101)(00)(01)(000)(100). Let M_N be the number of phrases produced by the algorithm for a string of length N. Aldous and Shields [3] proved that for the symmetric alphabet $(M_N - EM_N)/\text{Var}\, M_N$ converges weakly to the standard normal distribution, where $EM_N \sim N/\log_2 N$ and $\text{Var}\, M_N = O(N/\log_2^3 N)$, however, the authors of [3] were not able to provide the coefficient at $N/\log_2^3 N$. It turns out that this coefficient is the same as the coefficient at N in the variance of the internal path length L_N, that is, $\text{Var}\, M_N \sim (C + \delta(\log_2 N))N/\log_2^3 N$. The details of the proof can be found in [11]. The main idea is that the number of phrases M_N can be expressed as

$$M_N = \max\left\{ M : \ L_M = \sum_{k=1}^{M} D_M^{(k)} \le N \right\}.$$

But this equation is known in the literature as the *renewal equation*. Billingsley (compare [6], Chapter 17, Theorem 17.3) proved that (for any dependent positive random variables) if $(L - EL_N)/\sqrt{\text{Var}\, L_N}$ converges weakly to the standard normal distribution $\mathcal{N}(0, 1)$, then

$$\frac{M_N - N/(EL_N/N)}{\sqrt{\text{Var}\, L_N}(EL_N/N)^{-3/2}} \to \mathcal{N}(0, 1).$$

Hence, by the standard uniform integrability arguments and by Theorem 4, we conclude that the variance of M_N becomes

$$\text{Var}\, M_N = \ \text{Var}\, L_N/\log_2^3 N \sim (C + \delta(\log_2 N))N/\log_2^3 N.$$

3. Analysis. As we already pointed out in §2, it is a nontrivial problem to find appropriate analytic continuations for the sequences of values $f(k)$ that occur in alternating sums (2.5). In order to illustrate our approach, we start with the easiest case, namely, the evaluation of the expectation l_N. From (2.4) we know

$$l_N = \sum_{k=2}^{N} \binom{N}{k} (-1)^k Q_{k-2}.$$

As in [8] we may rewrite $f(k) = Q_{k-2}$ as

$$Q_{k-2} = \frac{Q_\infty}{Q(2^{2-k})},$$

where

$$Q(x) = \prod_{i \ge 1} \left(1 - \frac{x}{2^i}\right) \quad \text{and} \quad Q_\infty = Q(1). \tag{3.1}$$

Therefore, we have the analytic continuation

$$f(z) = \frac{Q_\infty}{Q(2^{2-z})}. \tag{3.2}$$

The main contribution to l_N is given by $\operatorname{Res}([N;z]f(z); z=1)$. We have with $u=z-1\to 0$

$$[N;z]\sim\frac{N}{u}\bigl(1+u(H_{N-1}-1)\bigr)$$

and

$$\frac{Q_\infty}{Q(2^{2-z})}\sim\frac{1}{Lu}\left(1+Lu\left(\frac{1}{2}-\alpha\right)\right)$$

(remember $L=\log 2$), since

$$\alpha=\left(\frac{Q_\infty}{Q(x)}\right)'\bigg|_{x=1}. \tag{3.3}$$

Therefore,

$$\operatorname{Res}([N;z]f(z);z=1)=\frac{N}{L}\left(H_{N-1}-1+L\left(\frac{1}{2}-\alpha\right)\right).$$

Using the well-known asymptotics for the *harmonic numbers* H_{N-1} we get the contribution (from $z=1$)

$$N\log_2 N+N\left(\frac{\gamma}{L}-\frac{1}{L}+\frac{1}{2}-\alpha\right)-\frac{1}{2L}+\mathcal{O}\left(\frac{1}{N}\right). \tag{3.4}$$

Besides $z=1$ we have with the same real part 1 the simple poles $z_k=1+\frac{2k\pi i}{L}$, $k\in\mathbf{Z}$, $k\neq 0$, with

$$\operatorname{Res}\bigl([N;z]f(z);z=z_k\bigr)=[N;z_k]\cdot\frac{1}{L},$$

so we get the contribution

$$\frac{N^{z_k}}{L}\Gamma(-z_k)-\frac{(z_k-1)z_kN^{z_k-1}\Gamma(-z_k)}{2L}+\mathcal{O}(N^{z_k-2}). \tag{3.5}$$

The reader should take notice of the fact that the first term in (3.5) gives the Fourier coefficients of $\delta_1(x)$ in Theorem 1.

The next relevant pole is $z=0$ and yields a contribution of

$$\log_2 N+\frac{\gamma}{L}+\frac{5}{2}-\alpha. \tag{3.6}$$

The poles $z=z_k-1$ yield a periodic contribution of order N^0 and so on.

Collecting all contributions gives the expansion (2.6) in Theorem 2.

Next we focus our attention on the *asymptotics of* u_N. In order to find an appropriate analytic continuation of $\hat{u}_k$ we rewrite the sums appearing in (2.12a) as follows:

$$\sum_{j=1}^{k-2}\frac{1}{2^j-1}=\alpha-\sum_{j\geq 1}\frac{1}{2^{k-2+j}-1},$$

$$\sum_{j=1}^{k-2}\frac{j}{2^j-1}=\sum_{j\geq 1}\frac{j}{2^j-1}-\sum_{j\geq 1}\frac{k-2+j}{2^{k-2+j}-1}.$$

Thus we may continue $\hat{u}_k$ via the function

$$\begin{aligned}\hat{u}(z) = \frac{2Q_\infty}{Q(2^{2-z})}\Bigg[& 4+\alpha-\sum_{j\geq 1}\frac{1}{2^{z-2+j}-1} \\ & -\sum_{j\geq 1}\frac{j}{2^j-1}+\sum_{j\geq 1}\frac{z-2+j}{2^{z-2+j}-1}-\frac{2z}{2^{z-2}-1}\Bigg].\end{aligned} \tag{3.7}$$

Now the main contribution to u_N in Lemma 1 originates from a second-order pole of $[N;\, z]\hat{u}(z)$ in $z=2$. Further contributions that are necessary for the evaluation of the variance come from first-order poles in $z = 2+\frac{2k\pi i}{L}$, $k\neq 0$, a third-order pole in $z=1$, as well as second-order poles in $z=1+\frac{2k\pi i}{L}$, $k\neq 0$. Collecting all the above-mentioned contributions we get the following expansion of u_N (in all the following formulas $\delta_i(x)$ stands for a continuous periodic function of period 1 and mean zero).

LEMMA 6.

$$\begin{aligned} u_N \sim{} & 4N^2\log_2 N + N^2\left(\frac{4(\gamma-1)}{L}-6-4\alpha+\delta_3(\log_2 N)\right) \\ & -N\log_2^2 N + 2N\log_2 N\cdot\left(\frac{2-\gamma}{L}+8+\alpha+\delta_4(\log_2 N)\right) \\ & +N\Bigg(-\frac{\gamma^2}{L^2}+\frac{4\gamma}{L^2}+\frac{12\gamma}{L}+\frac{2\alpha\gamma}{L}-\frac{\pi^2}{6L^2}-\frac{4}{L^2}-\frac{10}{L}-\frac{2\alpha}{L} \\ & -\alpha^2+\beta-11\alpha-2\beta_1+\frac{133}{6}+\delta_5(\log_2 N)\Bigg)+\mathcal{O}(\log^2 N)\end{aligned}$$

with $\beta=\sum_{k\geq 1}1/(2^k-1)^2$, L, α, and β_1 *as in Theorem* 4.

As already mentioned in §2,

$$v_N = 4\binom{N}{2}-4l_N,$$

so the *asymptotics of* v_N are given by

$$\begin{aligned} v_N \sim{} & 2N^2-4N\log_2 N \\ & +4N\left(-1+\alpha-\frac{\gamma}{L}+\frac{1}{L}-\delta_1(\log_2 N)\right)+\mathcal{O}(\log N).\end{aligned} \tag{3.8}$$

The most challenging task is to find an appropriate *analytic continuation of* $\hat{w}(z)$.

From (2.12c) we have

$$\hat{w}_{k+1} = -Q_{k-1}\sum_{j=4}^{k}\frac{\xi(j+1)}{2^{j-1}Q_{j-1}} \tag{3.9}$$

with

$$\xi(j+1) = \sum_{i=2}^{j-2}\binom{j}{i}Q_{i-2}Q_{j-2-i}. \tag{3.10}$$

For the following the reader should note that $\xi(j+1)\sim Q_\infty^2 2^j$. We start by rewriting (3.9) in the following manner:

With $\eta(j+1) = \xi(j+1) - Q_\infty^2 2^j$ we have

$$\begin{aligned}
-\frac{\hat{w}_{k+1}}{Q_{k-1}} &= \sum_{j=4}^{k} \frac{\eta(j+1) + Q_\infty^2 2^j}{2^{j-1} Q_{j-1}} \\
&= \sum_{j\ge 4} \frac{\eta(j+1)}{2^{j-1}Q_{j-1}} - \sum_{j\ge k+1} \frac{\eta(j+1)}{2^{j-1}Q_{j-1}} + 2Q_\infty(k-3) \qquad (3.11)\\
&\quad +2Q_\infty^2\left(\sum_{j\ge 4}\left(\frac{1}{Q_{j-1}} - \frac{1}{Q_\infty}\right) - \sum_{j\ge k+1}\left(\frac{1}{Q_{j-1}} - \frac{1}{Q_\infty}\right)\right).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\hat{w}_{k+1} = Q_{k-1}\Bigg[&-2Q_\infty(k-3) + \sum_{j\ge 0} \frac{\eta(j+k+2)}{2^{j+k}Q_{j+k}} - \sum_{j\ge 3}\frac{\eta(j+2)}{2^j Q_j} \\
&+2Q_\infty^2\left(\sum_{j\ge 0}\left(\frac{1}{Q_{j+k}} - \frac{1}{Q_\infty}\right) - \sum_{j\ge 3}\left(\frac{1}{Q_j} - \frac{1}{Q_\infty}\right)\right)\Bigg].
\end{aligned}$$

Since all involved series are now absolutely convergent, we may add them term-by-term and get

$$\hat{w}_{k+1} = Q_{k-1}\left[-2Q_\infty k + \frac{\xi(k+2)}{2^k Q_k} + \frac{\xi(k+3)}{2^{k+1}Q_{k+1}} + \sum_{j\ge 2}\left(\frac{\xi(k+j+2)}{2^{k+j}Q_{k+j}} - \frac{\xi(j+2)}{2^j Q_j}\right)\right].$$

From this, the representation for $\hat{w}(z+1)$ as in (2.14) is immediate, provided we have an appropriate interpretation for $\xi(z+1)$. This will be our next goal. The following well-known partition identities of Euler are our basic tool:

$$\frac{1}{Q(t)} = \prod_{n\ge 1}\frac{1}{(1-t2^{-n})} = \sum_{n\ge 0}\frac{t^n}{2^n Q_n} \qquad (3.12)$$

and

$$Q(t) = \prod_{n\ge 1}\left(1 - \frac{t}{2^n}\right) = \sum_{n\ge 0} a_{n+1} t^n \qquad (3.13)$$

with

$$a_{n+1} = (-1)^n 2^{-\binom{n+1}{2}}/Q_n.$$

Using (3.2) and (3.12) we have

$$\begin{aligned}
\xi(N+1) &= \sum_{k=2}^{N}\binom{N}{k}\frac{Q_\infty}{Q(2^{2-k})}\frac{Q_\infty}{Q(2^{2+k-N})} \\
&= Q_\infty^2 \sum_{i,j\ge 0}\frac{2^{i+j}}{Q_i Q_j}\sum_{k=2}^{N-2}\binom{N}{k}(2^{-i})^k(2^{-j})^{N-k},
\end{aligned}$$

where the innermost sum is now

$$(2^{-i} + 2^{-j})^N - 2^{-iN} - 2^{-jN} - N2^{-i(N-1)-j} - N2^{-i-j(N-1)}.$$

The last expression for $\xi(N+1)$ is symmetric in i and j. However, it turns out that for the purpose of finding an analytic continuation $\sum_{i,j\geq 0}$ should be rewritten as $-\sum_{j=i}+2\sum_{j\geq i\geq 0}$. Writing $j=i+h$ in the second sum we get

$$\begin{aligned}\xi(N+1)&=-(2^N-2-2N)\sum_{i\geq 0}\frac{Q_\infty^2}{Q_i^2}2^{i(2-N)}\\&+2\sum_{i,h\geq 0}\frac{Q_\infty^2}{Q_iQ_{i+h}}2^{i(2-N)}2^h\Big[(1+2^{-h})^N-1-N2^{-h}\Big]\\&-2\sum_{i,h\geq 0}\frac{Q_\infty^2}{Q_iQ_{i+h}}2^{i(2-N)}2^{h(1-N)}\\&-2N\sum_{i,h\geq 0}\frac{Q_\infty^2}{Q_iQ_{i+h}}2^{i(2-N)}2^{h(2-N)}.\end{aligned}\tag{3.14}$$

In expression (3.14) N can be replaced by z, yielding a meromorphic function, since all series converge uniformly. However, we are able to simplify $\xi(z+1)$ in the following way. Consider, for example, the last term in (3.14):

$$2z\sum_{i,h\geq 0}\frac{Q_\infty^2}{Q_iQ_{i+h}}2^{(i+h)(2-z)}=2z\sum_{i,h\geq 0}\frac{Q_\infty Q(2^{-i-h})}{Q_i}2^{(i+h)(2-z)},$$

which is by Euler's identity (3.13)

$$=2z\sum_{r\geq 0}a_{r+1}\sum_{i\geq 0}\frac{Q_\infty}{Q_i}(2^{-r+2-z})^i\sum_{h\geq 0}(2^{-r+2-z})^h$$

and by Euler's identity (3.12)

$$=2z\sum_{r\geq 0}a_{r+1}\frac{Q_\infty}{Q(2^{3-z-r})}\cdot\frac{1}{1-2^{2-z-r}}.$$

Rewriting the other terms from (3.14) in a similar way, especially using

$$(1+2^{-h})^z-1-z2^{-h}=\sum_{k\geq 2}\binom{z}{k}2^{-hk}$$

for the second term, we finally get (2.15).

Our next task is to investigate the poles of $[N;z]\hat{w}(z)$ different from $z=5,6,\ldots,N$.

From (3.11) we see that $\hat{w}(4)=\hat{w}(3)=0$ (observe that $\xi(4)=0$), so that the first poles occur with real part 2. In order to determine the residues of $[N;z]\hat{w}(z)$ in $z=2$ (respectively, $z=1$) we need the local behavior of $\hat{w}(z)$. Because of (2.14) this behavior will depend on the behavior of $\sigma(z):=\xi(z)/2^{z-2}Q_{z-2}$ near $z=2,3,\ldots$. From (2.15) we see that $\xi(z)$ has second-order poles for $z=2$ and $z=3$ and is analytic for $z=4,5,\ldots$. Since $[N;z]Q_{z-2}$ has already a second-order pole for $z=1$, it will be necessary to expand $\sigma(z)$ near $z=2,3,\ldots$ up to the linear terms. In particular, the reader should note that *all* the derivatives $\sigma'(z)$ for $z=4,5,\ldots$ will occur in $\operatorname{Res}\big([N;z]\hat{w}(z);z=1\big)$. This is the main reason that the constant C in the final result is rather a complicated one.

We start with the expansion of $\sigma(z)$ about $z = 3$. Let $u = z - 3$; then we find from (2.15) after laborious computations:

$$\begin{aligned}\sigma(3+u) \sim & -\frac{4}{L^2u^2} + \frac{1}{u}\left(\frac{6}{L} - \frac{2}{L^2}\right)\\ & +u^0\left(\frac{16}{3} + \frac{5}{L} + \frac{2\beta_2}{L} + 2\sum_{r\geq 2}\frac{b_{r+1}}{1-2^{-r}}\left(1 - \frac{2}{1-2^{-r}}\right)\right)\\ & +u^1\left(-\frac{223}{18}L - \frac{23}{6} - 3C_1 + \frac{C_2}{L} - 2C_3 + 2\sum_{r\geq 2}\frac{b_{r+1}}{1-2^{-r}}\right.\\ & \cdot\left\{1 - 3L + L\sum_{i=2}^{r-1}\frac{1}{2^i-1} + \frac{2^{-1-r}L}{(1-2^{-1-r})^2} + \frac{2L-1}{1-2^{-r}}\right.\\ & \left.\left. -\frac{2L}{1-2^{-r}}\sum_{i=2}^{r-1}\frac{1}{2^i-1} + \frac{2^{1-r}L}{(1-2^{-r})^2} + D_{r,1}\right\}\right)\end{aligned} \tag{3.15}$$

where

$$\begin{aligned}\beta_2 &= 2\sum_{k\geq 2}\frac{(-1)^k}{(k+1)k(k-1)(2^k-1)},\\ C_1 &= \sum_{h\geq 0}2^h\Big[(1+2^{-h})^2\log(1+2^{-h}) - 2^{-h}\Big],\\ C_2 &= \sum_{h\geq 0}2^h(1+2^{-h})^2\log^2(1+2^{-h}),\\ C_3 &= \sum_{h\geq 0}\Big[(1+2^{-h})^2\log(1+2^{-h}) - 2^{-h}\Big],\\ D_{r,1} &= \sum_{h\geq 0}2^{h(1-r)}\Big[(1+2^{-h})^2\log(1+2^{-h}) - 2^{-h}\Big].\end{aligned}$$

The constant in the u^0-term can be simplified according to the remarkable identity

$$\sum_{r\geq 1}\frac{b_{r+1}}{1-2^{-r}}\left(1 - \frac{2}{1-2^{-r}}\right) = \sum_{j\geq 1}\frac{2^j}{(2^j-1)^2} = \alpha + \beta. \tag{3.16}$$

For the proof of (3.16) we observe that the left-hand side equals

$$\sum_{r\geq 1}a_{r+1}Q_{r-1} - 2\sum_{r\geq 1}a_{r+1}\frac{Q_{r-1}}{1-2^{-r}}.$$

Using (3.12) and (3.13) this expression becomes

$$\begin{aligned}&\sum_{i\geq 0}Q(2^{-i})\left(Q(2^{-i}) - 1\right) - 2\sum_{i,k\geq 0}Q(2^{-i})\left(Q(2^{-i-k}) - 1\right)\\ &= -\left(\sum_{i\geq 0}\left(Q(2^{-i}) - 1\right)\right)^2 - \sum_{j\geq 0}(2j+1)\left(Q(2^{-j}) - 1\right)\\ &= -E_1^2 - 2E_2 - E_1,\end{aligned}$$

where

$$E_1 = \sum_{r\geq 1} \frac{a_{r+1}}{1-2^{-r}} \quad \text{and} \quad E_2 = \sum_{r\geq 1} a_{r+1} \frac{2^{-r}}{(1-2^{-r})^2}.$$

Now we observe

$$E_1 = \lim_{t\to 2} \left[\frac{Q_\infty}{Q(t)} - \frac{1}{1-t/2} \right] = - \left(\frac{Q_\infty}{Q(t)} \right)' \Bigg|_{t=1} = -\alpha$$

and

$$E_2 = 2 \lim_{t\to 2} \left[\left(\frac{Q_\infty}{Q(t)} \right)' - \frac{1/2}{(1-t/2)^2} \right] = -\frac{1}{2} \left(\frac{Q_\infty}{Q(t)} \right)'' \Bigg|_{t=1} = -\frac{\alpha^2+\beta}{2}$$

and get the right-hand side of (3.16).

The expansion of $\sigma(z)$ about $z=2$ reads with $u = z-2$:

$$\begin{aligned} \sigma(2+u) \sim {} & \frac{4}{L^2u^2} + \frac{1}{u}\left(\frac{2}{L} + \frac{2}{L^2}\right) \\ & + u^0\left(-\frac{16}{3} - \frac{6}{L} - \frac{2}{L}(C_4+C_6) + 2E_3\right) \\ & + u^1\Bigg(\frac{L}{6} - \frac{55}{6} - 3C_4 - \frac{C_5}{L} + C_6 - \frac{C_7}{L} \\ & + 2\sum_{r\geq 2} \frac{b_{r+1}}{(1-2^{1-r})(1-2^{-r})} \\ & \cdot\Bigg\{ D_{r,2} + 1 - 2L + L\sum_{i=1}^{r-2}\frac{1}{2^i-1} + \frac{L-1}{1-2^{1-r}} \\ & + L\Bigg(\frac{1}{(1-2^{-r})^2} - \frac{1}{1-2^{-r}}\sum_{i=1}^{r-2}\frac{1}{2^i-1} + \frac{2^{1-r}}{(1-2^{1-r})^2} \\ & - \frac{1}{1-2^{1-r}}\sum_{i=1}^{r-2}\frac{1}{2^i-1}\Bigg)\Bigg\}\Bigg) \end{aligned} \tag{3.17}$$

where

$$\begin{aligned} C_4 &= \sum_{h\geq 0} 2^h \left[(1+2^{-h})\log(1+2^{-h}) - 2^{-h}\right], \\ C_5 &= \sum_{h\geq 0} 2^h (1+2^{-h})\log^2(1+2^{-h}), \\ C_6 &= \sum_{h\geq 0} \left[(1+2^{-h})\log(1+2^{-h}) - 2^{-h}\right], \\ C_7 &= \sum_{h\geq 0} (1+2^{-h})\log^2(1+2^{-h}), \\ D_{r,2} &= \sum_{h\geq 0} 2^{h(1-r)} \left[(1+2^{-h})\log(1+2^{-h}) - 2^{-h}\right], \\ E_3 &= \sum_{r\geq 2} \frac{b_{r+1}}{(1-2^{1-r})(1-2^{-r})}\left(1 - \frac{1}{1-2^{-r}} - \frac{1}{1-2^{1-r}}\right). \end{aligned}$$

For later simplifications we note that

$$-\frac{2}{L}(C_4+C_6) = -8 + \frac{3}{L} - \frac{2\beta_2}{L} \tag{3.18}$$

and

$$E_3 = 2 - \alpha - \beta, \tag{3.19}$$

where (3.18) follows from the expansion of the logarithm and (3.19) by partial fraction decomposition and rearrangements of the sums.

We finally note that

$$C_5 + C_7 = C_2. \tag{3.20}$$

Next we discuss $\sigma(z)$ for z close to $j = 4, 5, \dots$.

$$\sigma(j+u) \sim \sigma(j) + \frac{u}{2^{j-2}Q_{j-2}}\left(\xi'(j) - L\xi(j) + L\xi(j)\sum_{k\geq 1}\frac{1}{2^{k+j-2}-1}\right). \tag{3.21}$$

From (3.21)

$$\begin{aligned}&\sum_{j\geq 4}\Bigl(\sigma(j+u)-\sigma(j)\Bigr)\\ &\quad\sim u\left(\sum_{j\geq 4}\frac{\xi'(j)-L\xi(j)}{2^{j-2}Q_{j-2}} + L\sum_{j\geq 4}\frac{\xi(j)}{2^{j-2}Q_{j-2}}\sum_{k\geq 1}\frac{1}{2^{k+j-2}-1}\right).\end{aligned} \tag{3.22a}$$

From (2.14) we find that the last expression equals

$$u\bigl(2\hat{w}'(3) + 2Q_\infty\bigr), \tag{3.22b}$$

where $\hat{w}'(3)$ may be computed from (2.15) to get the constant from Lemma 5.

Regarding (3.15), (3.17), and (3.22a) we find that $[N; z]\hat{w}(z)$ has third-order poles in $z = 2$ and $z = 1$ and second-order poles in $z_k = 2 + \frac{2k\pi i}{L}$, $k \in \mathbb{Z}$, $z \neq 0$, as well as in $z_k - 1$.

Our local expansions allow (after some lengthy but straightforward computations) to find the following asymptotic behavior of w_N:

LEMMA 7.

$$\begin{aligned}w_N = {}& N^2\log_2^2 N + N^2\log_2 N\cdot\left(-3 - \frac{2}{L} + \frac{2\gamma}{L} - 2\alpha + \delta_6(\log_2 N)\right)\\ &+N^2\biggl(\frac{1}{3} + \alpha^2 + 3\alpha + \frac{2\alpha}{L} - \frac{3\gamma}{L} - \frac{2\alpha\gamma}{L} - \frac{\beta_2}{L}\\ &+\frac{2}{L} + \frac{2}{L^2} + \frac{\gamma^2}{L^2} + \frac{\pi^2}{6L^2} - \frac{2\gamma}{L^2} + \delta_7(\log_2 N)\biggr)\end{aligned}$$

$$+3N\log_2^2 N + N\log_2 N\cdot\left(-\frac{7}{L}-3-6\alpha-\frac{10\gamma}{L}+\delta_8(\log_2 N)\right)$$
$$+N\left(-22-\frac{41}{6L}+\frac{\beta_2}{L}-\frac{3\gamma}{L}-\frac{7\gamma}{L^2}+2\alpha-\beta+\frac{7\alpha}{L}+3\alpha^2-\frac{6\alpha\gamma}{L}+\frac{3\gamma^2}{L^2}\right.$$
$$+\frac{\pi^2}{2L^2}+\frac{6}{L^2}-\frac{2}{L}\sum_{k\geq 3}\frac{(-1)^{k+1}(k-5)}{(k+1)k(k-1)(2^k-1)}+\frac{2}{L}\sum_{r\geq 1}b_{r+1}$$
$$\cdot\left(\frac{L(1-2^{-r+1})/2-1}{1-2^{-r}}-\sum_{k\geq 2}\frac{(-1)^{k+1}}{k(k-1)(2^{r+k}-1)}\right)$$
$$\left.+\frac{2}{L}\hat{w}'(3)+\delta_9(\log_2 N)\right)+\mathcal{O}\left(\frac{\log^2 N}{N}\right)$$

with $L, \alpha, \beta, b_{r+1}$ *as in Theorem* 4 (*respectively, Lemma* 6) *and* β_2 *from* (3.15).

It remains to combine the previous results to get an asymptotic expansion for

$$\operatorname{Var} L_N = u_N + v_N + w_N + l_N - l_N^2. \tag{3.23}$$

We start with an important observation concerning leading terms formed by periodic fluctuations of mean zero.

Let us assume that, at any stage, we are able to prove

$$\operatorname{Var} L_N = \delta_{10}(\log_2 N)\cdot N^\mu \log^\nu N + R_N, \tag{3.24}$$

where $\delta_{10}(x)$ is continuous and periodic with period 1 and mean zero and $R_N = o(N^\mu \log^\nu N)$. We claim that $\delta_{10}(x)$ must vanish identically under these conditions:

Let us assume $\delta_{10}(x) \not\equiv 0$. Then, since $\delta_{10}(x)$ is continuous with mean 0, there esists an $\epsilon > 0$ and an interval, say $[a, b] \subseteq [0, 1]$, such that $\delta_{10}(x) < -\epsilon$ for $x \in [a, b]$. Since $\log_2 N$ is dense modulo 1, $\operatorname{Var} L_N$ would be negative for an infinity of values, an obvious contradiction.

In other words: From (3.24) we may deduce that

$$\operatorname{Var} L_N \sim R_N, \qquad N \to \infty, \tag{3.25}$$

so, in order to prove that $\operatorname{Var} L_N = \mathcal{O}(N)$ we need not collect explicitly the fluctuating contributions of mean zero.

Observing these comments we easily find that all terms of order $N^2\log^2 N$, $N^2\log N$, $N\log^2 N$, and $N\log N$ in $\operatorname{Var} L_N$ cancel. The coefficient of N^2 is of a more delicate nature. The reader should note that the coefficient of N^2 in l_N^2 will contain the square δ_1^2 of the periodic fluctuation δ_1 from Theorem 2 and that the mean $[\delta_1^2]_0$ of δ_1^2 will *not* be zero. Therefore, we have to extract this term to end up with a fluctuation of mean zero and get for the coefficient of N^2 in $\operatorname{Var} L_N$ the expression

$$\frac{1}{L^2}+\frac{\pi^2}{6L^2}-\frac{1}{L}-\frac{\beta_2}{L}-\frac{47}{12}-[\delta_1^2]_0+\delta_{11}(\log_2 N). \tag{3.26}$$

The following lemma is crucial now.

LEMMA 8.

$$[\delta_1^2]_0 = \frac{1}{L^2}\sum_{k\neq 0}\left|\Gamma\left(-1-\frac{2k\pi i}{\log 2}\right)\right|^2 = \frac{1}{L^2}+\frac{\pi^2}{6L^2}-\frac{1}{L}-\frac{\beta_2}{L}-\frac{47}{12}.$$

Sketch of Proof[1]. The proof heavily relies on the following two series transformation results due to Ramanujan (compare [5]). The first is

$$\begin{aligned}(3.27)\qquad &\alpha^{-N}\left(\frac{1}{2}\zeta(2N+1)+\sum_{k\geq 1}\frac{k^{-2N-1}}{e^{2\alpha k}-1}\right)\\ &=(-\beta)^{-N}\left(\frac{1}{2}\zeta(2N+1)+\sum_{k\geq 1}\frac{k^{-2N-1}}{e^{2\beta k}-1}\right)\\ &-2^{2N}\sum_{k=0}^{N+1}(-1)^k\frac{B_{2k}}{(2k)!}\frac{B_{2N+2-2k}}{(2N+2-2k)!}\alpha^{N+1-k}\beta^k.\end{aligned}$$

Here and in the next identity, α and β have to be positive numbers with $\alpha\beta=\pi^2$, $\zeta(s)$ is the Riemann ζ-function; N has to be a positive integer, and B_n indicates the nth Bernoulli number defined by

$$\frac{z}{e^z-1}=\sum_{n\geq 0}B_n\frac{z^n}{n!}.$$

The second identity used in the proof is

$$(3.28)\qquad \sum_{k\geq 1}\frac{1}{k(e^{2\alpha k}-1)}-\frac{1}{4}\log\alpha+\frac{\alpha}{12}=\sum_{k\geq 1}\frac{1}{k(e^{2\beta k}-1)}-\frac{1}{4}\log\beta+\frac{\beta}{12}.$$

In fact, (3.28) is equivalent to a transformation result on Dedekind's η-function (compare [4])

$$(3.29)\qquad \eta(\tau)=e^{\pi i\tau/12}\prod_{n\geq 1}\left(1-e^{2\pi in\tau}\right),\qquad \Im(\tau)>0,$$

namely,

$$(3.30)\qquad \eta\left(-\frac{1}{\tau}\right)=(-i\tau)^{1/2}\cdot\eta(\tau),\qquad \Im(\tau)>0.$$

(This is a special instance of Dedekind's famous result on the behavior of η under a transformation of the modular group.) □

The consequences of Lemma 8 are twofold. On the one hand, we find from (3.26) that the N^2-term in Var L_N cancels, so that Var $L_N=\mathcal{O}(N)$. On the other hand we may use the identity to express β_2 by the other terms occurring in Lemma 8, including $[\delta_1^2]_0$, which yields the final form of the constant C in Theorem 2. $[\delta_1\delta_2]_0$ is the mean of $\delta_1(x)\delta_2(x)$, originating from l_N^2, which has to be extracted to get a fluctuation $\delta(x)$ of mean zero.

4. Concluding remarks. We would like to point out some final remarks concerning our analysis.

(i) The occurrence of the finite products Q_k gives rise to use results from the *theory of partitions*, especially the Euler product identities (3.12) and (3.13).

[1] A full proof of Lemma 8 is long and difficult and included in [13].

(ii) A periodic fluctuation $\delta_1(x)$ that has mean zero and very small amplitude may be safely neglected for practical purposes as long as we are only interested in the *expectation.* In order to establish the correct order of the *variance* it is of vital importance to study the behavior of $\delta_1(x)$, especially the mean of $\delta_1^2(x)$.

(iii) The predicted value $0.26600\ldots \cdot N$ matches perfectly with the values obtained by *computer simulations.*

(iv) As we mentioned in the introduction, with this paper we achieved our goal of obtaining second-order properties for the three digital tree search structures, namely, *Tries, Patricia Tries*, and *Digital Search Trees* in the symmetric case. In particular, we are now able to compare the variances of the path lengths in such trees. Furthermore, we note that the analysis of the variances is a key step toward getting the respective limiting distributions (compare [3] or [10]).

(v) Our methodology does not easily extend to the asymmetric Bernoulli model. The difficulty lies in the analytical continutation of w_N. We conjecture that in the asymmetric case the variance is of order $n \log n$, but it might be difficult to obtain the coefficient at $n \log n$.

Acknowledgment. The authors would like to thank R. F. Tichy for some helpful remarks.

REFERENCES

[1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, Dover, New York, 1970.

[2] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison Wesley, Reading, MA, 1983.

[3] D. Aldous and P. Shields, *A diffusion limit for a class of random-growing binary trees*, Probab. Theory Related Fields, 79 (1988), pp. 509–542.

[4] T. M. Apostol, *Modular Functions and Dirichlet Series in Number Theory*, Springer, New York, 1976.

[5] B. C. Berndt, *Ramanujan's Notebooks Part* II, Springer, New York, 1989.

[6] P. Billingsley, *Convergence of Probability Measures*, John Wiley and Sons, New York, 1968.

[7] E. G. Coffman, Jr. and J. Eve, *File structures using hashing functions*, Comm. Appl. Comput. Mach., 13 (1970), pp. 427–436.

[8] P. Flajolet and R. Sedgewick, *Digital search trees revisited*, SIAM J. Comput., 15 (1986), pp. 748–767.

[9] G. Gonnet, *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA, 1983.

[10] P. Jacquet and M. Regnier, *Normal Limiting Distributions for the Size and the External Path Length of Tries*, INRIA Tech. report, 827, 1988.

[11] P. Jacquet and W. Szpankowski, *On the Lempel-Ziv Parsing Algorithm and Its Digital Tree Representation*, INRIA Tech. report, 1833, 1992.

[12] P. Kirschenhofer and H. Prodinger, *Further results on digital search trees*, Theoret. Comput. Sci., 58 (1988), pp. 143–154.

[13] ———, *On some applications of formulae of Ramanujan in the analysis of algorithms*, Mathematika, 38 (1991), pp. 14–33.

[14] P. Kirschenhofer, H. Prodinger, and W. Szpankowski, *On the variance of the external path length in a symmetric digital tree*, Discrete Appl. Math., 25 (1989), pp. 129–143.

[15] ———, *On the balance property of Patricia trees: External path length viewpoint*, Theoret. Comput. Sci., 68 (1989), pp. 1–17.

[16] ———, *Digital search trees—further results on a fundamental data structure*, Proceedings of IFIP 89, G. X. Ritter, ed. (1989), pp. 443–447.

[17] D. E. Knuth, *The Art of Computer Programming, Vol.* 3, Addison-Wesley, Reading, MA, 1973.

[18] A. G. Konheim and D. J. Newman, *A note on growing binary trees*, Discrete Mathematics, 4 (1973), pp. 57–63.

[19] A. Lempel and J. Ziv, *On the complexity of finite sequences*, IEEE Trans. Inform. Theory, 22 (1976), pp. 75–81.

[20] G. Louchard, *Exact and asymptotic distributions in digital and binary search trees*, RAIRO Theoret. Inform. Appl., 21 (1987), pp. 479–495.

[21] H. Mahmoud, *Evolution of Random Search Trees*, John Wiley and Sons, New York, 1992.

[22] P. Mathys and P. Flajolet, *Q-ary collision resolution algorithms in random-access systems with free and blocked channel access*, IEEE Trans. Inform. Theory, 31 (1985), pp. 217–243.

[23] N. E. Nörlund, *Vorlesungen über Differenzenrechnung*, Chelsea, New York, 1954.

[24] B. Pittel, *Paths in a random digital tree: limiting distributions*, Adv. Appl. Probab., 18 (1986), pp. 139–155.

[25] M. Régnier and P. Jacquet, *New results on the size of tries*, IEEE Trans. Inform. Theory, 35 (1989), pp. 203–205.

[26] J. Riordan, *Combinatorial Identities*, John Wiley and Sons, New York, 1968.

[27] W. Schachinger, *Modulfunktionen und Analyse von Datenstrukturen*, Diplomarbeit, TU Vienna, 1990.

[28] R. Sedgewick, *Algorithms*, Second ed., Addison-Wesley, Reading, MA, 1988.

[29] W. Szpankowski, *Some results on V-ary asymmetric trees*, J. Algorithms, 9 (1988), pp. 224–244.

[30] ———, *A characterization of digital search trees from the successful search viewpoint*, Theor. Comput. Sci., 85 (1991), pp. 117–134.

[31] ———, *Patricia trees again revisited*, J. Assoc. Comput. Mach., 37 (1990), pp. 691–711.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 617–632, June 1994

010

IMPROVED APPROXIMATION ALGORITHMS FOR SHOP SCHEDULING PROBLEMS*

DAVID B. SHMOYS†, CLIFFORD STEIN‡, AND JOEL WEIN§

Abstract. In *the job shop scheduling* problem, there are m machines and n jobs. A job consists of a sequence of operations, each of which must be processed on a specified machine, and the aim is to complete all jobs as quickly as possible. This problem is strongly $\mathcal{NP}$-hard even for very restrictive special cases. The authors give the first randomized and deterministic polynomial-time algorithms that yield polylogarithmic approximations to the optimal length schedule. These algorithms also extend to the more general case where a job is given not by a linear ordering of the machines on which it must be processed but by an arbitrary partial order. Comparable bounds can also be obtained when there are m' types of machines, a specified number of machines of each type, and each operation must be processed on one of the machines of a specified type, as well as for the problem of scheduling unrelated parallel machines subject to chain precedence constraints.

Key words. scheduling, approximation algorithms

AMS subject classifications. 68A10, 68Q25, 90B35, 68R99

1. Introduction.

In the *job shop scheduling* problem we are given m machines and n jobs. A job consists of a sequence of operations, each of which must be processed on a specified machine; a job may have more than one operation on a given machine. The operations of a job must be processed in the order specified by the sequence, subject to the constraint that on each machine at most one job is scheduled at any point in time. We wish to produce a schedule of jobs on machines that minimizes $C_{\max}$, the time when all jobs have completed processing. This problem is strongly $\mathcal{NP}$-hard; furthermore, except for the cases when there are two jobs or when there are two machines *and* each job has at most two operations, essentially all special cases of this problem are $\mathcal{NP}$-hard, and typically strongly $\mathcal{NP}$-hard [6], [7]. For example, it is $\mathcal{NP}$-hard even if there are three machines, three jobs, and each operation is of unit length; note that in this case we can think of the input length as μ, the maximum number of operations in a job. In addition to these theoretical results, the job shop problem is also one of the most notoriously difficult $\mathcal{NP}$-hard optimization problems in terms of practical computation, with even very small instances being difficult to solve exactly. A classic single instance of this problem involving only 10 jobs, 10 machines, and 100 operations, which was published in 1963, remained unsolved for 23 years despite repeated attempts to find an optimal solution [7]. Furthermore, several benchmark instances with 15 jobs, 15 machines, and 225 operations are too hard to be solved by known methods and were posed as open problems by Applegate and Cook [1].

In this paper we will focus on obtaining approximation algorithms for the job shop problem, and will evaluate these algorithms in terms of their performance guarantee, or in other words, their worst-case relative error. Let $C^*_{\max}$ be the maximum completion time of a job in

*Received by the editors February 17, 1992; accepted for publication (in revised form) March 11, 1993. This research was partially supported by National Science Foundation Presidential Young Investigator award CCR-89-96272 with matching support from UPS and Sun; by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS89-20550, and by Defense Advanced Research Projects Agency contract N00014-89-J-1988. A preliminary version of this paper appeared in the Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, January 1991.

†School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853 (shmoys@cs.cornell.edu).

‡Department of Mathematics and Computer Science, 6211 Sudilcoff Laboratory, Dartmouth College, Hanover, New Hampshire 03755-3551 (cliff@bondcliff.dartmouth.edu). Additional support provided by an AT&T graduate fellowship.

§Department of Computer Science, Polytechnic University, Five MetroTech Center, Brooklyn, New York 11201 (wein@mem.poly.edu). Additional support provided by an Army Research Office graduate fellowship.

the optimal solution. If a polynomial-time algorithm always delivers a solution of maximum completion time of at most $\rho C^*_{\max}$, then we shall call it a ρ-approximation algorithm. The main result of this paper is the first randomized polynomial-time polylogarithmic approximation algorithm for the job shop scheduling problem.

THEOREM 1.1. *There exists a polynomial-time randomized algorithm for the job shop scheduling problem, that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max}\right).$$

We formally define the job shop problem as follows. We are given a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of machines, a set $\mathcal{J} = \{J_1, \ldots, J_n\}$ of jobs, and a set $\mathcal{O} = \{O_{ij} | i = 1, \ldots, \mu_j, j = 1, \ldots, n\}$ of operations, where κ_{ij} indexes the machine which must process operation O_{ij}. Thus m is the number of machines, n is the number of jobs, μ_j is the number of operations of job J_j, and $\mu = \max_j \mu_j$. Operation O_{ij} is the ith operation of J_j; it requires processing time on a given machine $M_k \in \mathcal{M}$, where $k = \kappa_{ij}$, for an uninterrupted period of a given length p_{ij}. (In other words, this is a *nonpreemptive* model; a model in which operations may be interrupted and resumed at a later time is called a *preemptive* model.) Each machine can process at most one operation at a time, and each job may be processed by at most one machine at a time. If the completion time of operation O_{ij} is denoted by C_{ij}, then the objective is to produce a schedule that minimizes the maximum completion time, $C_{\max} = \max_{i,j} C_{ij}$; the optimal value is denoted by $C^*_{\max}$.

Note that there are two very easy lower bounds on the length of an optimum schedule. Since each job must be processed, $C^*_{\max}$ must be at least the maximum total length of any job, $\max_{J_j} \sum_i p_{ij}$, which we shall call the *maximum job length* of the instance, and denote by $P_{\max}$. Furthermore, each machine must process all of its operations, and so $C^*_{\max}$ must be at least $\max_{M_k} \sum_{\kappa_{ij}=k} p_{ij}$, which we shall call the *maximum machine load* of the instance, and denote by $\Pi_{\max}$.

Our work is based on two very different approaches to the job shop problem. One approach is a geometric approach to shop scheduling, while the other is a randomized approach that finds its genesis in problems of packet routing. We briefly review both approaches here.

The best approximation algorithms to date for job shop scheduling have primarily appeared in the Soviet literature and are based on a beautiful connection to geometric arguments. This approach was independently discovered by Belov and Stolin [3] and Sevast'yanov [18], as well as by Fiala [4]. This approach typically produces schedules for which the length can be bounded by $\Pi_{\max} + q(m, \mu)p_{\max}$, where $q(\cdot, \cdot)$ is a polynomial, and $p_{\max} = \max_{ij} p_{ij}$ is the maximum operation length. For the job shop problem, Sevast'yanov [16], [17] gave a polynomial-time algorithm that delivered a schedule of length at most $\Pi_{\max} + O(m\mu^3)p_{\max}$. The bounds obtained in this way do not give good worst-case relative error bounds. Even for the special case of the *flow shop problem*, where each job has a single operation on each machine and for each job the operations must be processed in the same order, the best known algorithms delivered solutions of length $\Omega(mC^*_{\max})$.

In a different vein, Leighton, Maggs, and Rao [8] have proposed the following model for the routing of packets in a network: Find paths for the packets and then schedule the transmission of the packets along these paths so that no two packets traverse the same edge simultaneously. The primary objective is to minimize the time by which all packets have been delivered to their destination.

It is easy to see that the problem considered by Leighton, Maggs, and Rao is simply the job shop scheduling problem with each processing time $p_{ij} = 1$. They also added the restriction that each path does not traverse any edge more than once, or in scheduling terminology, each job has at most one operation on each machine. This restriction of the job shop problem remains (strongly) $\mathcal{NP}$-hard. The main result of Leighton, Maggs, and Rao was to show that for their special case of the job shop problem, there always exists a schedule of length $O(\Pi_{\max} + P_{\max})$. Unfortunately, this is not an algorithmic result, as it relies on a nonconstructive probabilistic argument based on the Lovász Local Lemma. They also obtained a randomized algorithm that delivers a schedule of length $O(\Pi_{\max} + P_{\max} \log n)$, with high probability. In this paper, we will show how their techniques can be generalized to handle the general job shop problem, as well as several related scheduling problems.

We also give a deterministic version of the job shop scheduling algorithm.

THEOREM 1.2. *There exists a deterministic polynomial-time algorithm for job shop scheduling that finds a schedule of length* $O(\log^2(m\mu)C^*_{\max})$.

This is the first polylogarithmic performance guarantee for a deterministic polynomial time approximation algorithm for job shop scheduling; no such algorithm was known even for the special case of flow shop scheduling. Note that if each job must be processed on each machine at most once, the factor of μ can be deleted for this, and all other performance guarantees in this paper. As a corollary, we also obtain a deterministic version of the randomized algorithm of Leighton, Maggs, and Rao. Our results rely on results of Raghavan and Thompson [14] and Raghavan [12] to approximate certain integer packing problems.

In contrast to this, the only "negative" result previously known for any shop scheduling problem is that the existence of a fully polynomial approximation scheme would imply that $\mathcal{P} = \mathcal{NP}$, due to the fact that these problems are strongly $\mathcal{NP}$-hard. Subsequent to our work, Williamson et al. [20] showed that the existence of a ρ-approximation algorithm for any shop scheduling problem with $\rho < 5/4$ would imply that $\mathcal{P} = \mathcal{NP}$.

Our techniques can also be made to apply to three important generalizations of the job shop problem. The first is *dag shop scheduling*, where each job consists of a set of operations on different machines that must be processed in an order consistent with a given partial order. (For job shop scheduling, this partial order is always a chain, while for flow shop the partial order is the same chain for all jobs.) Note that we still require that no two operations of the same job can be processed simultaneously. One can further generalize the problem to the situation where, rather than having m different machines, there are m' types of machines, and for each type, there are a specified number of identical machines; each operation, rather than being assigned to one machine, may be processed on any machine of the appropriate type. These problems have significant practical importance, since in real-world shops we would expect that a job need not follow a strict total order and that the shop would have more than one copy of many of their machines. Finally, a further generalization of this problem is the problem of scheduling on unrelated parallel machines subject to chain precedence constraints. In this problem, each job now consists of a single operation which may be scheduled on any machine, but its processing time depends on the machine on which it is scheduled. There is a set of disjoint chain precedence constraints that further restricts the order in which the jobs may be scheduled.

We also give some extensions of these results, including an $\mathcal{RNC}$ approximation algorithm for each scheduling model mentioned above, and a $(2 + \epsilon)$-approximation algorithm for the job shop scheduling problem with a fixed number of machines.

While all of the algorithms that we give are polynomial-time, they are all rather inefficient. Most rely on the algorithms of Sevast'yanov; for example, his algorithm for the job shop scheduling problem takes $O((\mu mn)^2)$ time. Furthermore, the deterministic versions rely on linear programming algorithms. As a result, we will not refer explicitly to running times throughout the remainder of this paper.

2. The basic algorithm.

In this section we extend the technique introduced by Leighton, Maggs, and Rao [8] of assigning random delays to jobs to the general case of non-preemptive job shop scheduling.

A valid schedule assigns at most one job to a particular machine at any time, and schedules each job on at most one machine at any time. Our approach will be to first create a schedule that obeys only the second constraint, and then build from this a schedule that satisfies both constraints and is not much longer. The outline of the strategy follows.

(1) Define the *oblivious* schedule, where each job starts running at time 0 and runs continuously until all of its operations have been completed. This schedule is of length $P_{\max}$, but there may be times when more than one job is assigned to a particular machine.

(2) Perturb this schedule by delaying the start of the first operation of each job by a random integral amount chosen uniformly in $[0, \Pi_{\max}]$. The resulting schedule, with high probability, has

$$O\left(\frac{\log(n\mu)}{\log\log(n\mu)}\right)$$

jobs assigned to any machine at any time.

(3) "Spread" this schedule so that at each point in time all operations currently being processed have the same size, and then "flatten" this into a schedule that has at most one job per machine at any time.

This strategy is very similar to the one used by Leighton, Maggs, and Rao for the special case of unit-length operations. Whereas Step 2 differs in only a few technical details, the essential difficulty in obtaining the generalization is in Step 3. For the analysis of Step 2, we assume that $p_{\max}$ is bounded above by a polynomial in n and μ; in the next section we will show how to remove this assumption. For simplicity, we shall assume that $n \geq m$; analogous bounds can be obtained when this is not true.

LEMMA 2.1. *Given a job shop instance in which $p_{\max}$ is bounded above by a polynomial in n and μ, the strategy of delaying each job an initial integral amount chosen randomly and uniformly from $[0, \Pi_{\max}]$ and then continuously processing its operations in sequence will yield an (invalid) schedule that is of length at most $\Pi_{\max} + P_{\max}$ and, with high probability, has*

$$O\left(\frac{\log(n\mu)}{\log\log(n\mu)}\right)$$

jobs scheduled on any machine during any unit of time.

Proof. Fix a time t and a machine M_i; let $p =$ Prob[at least τ units of processing are scheduled on machine M_i at time t]. There are at most $\binom{\Pi_{\max}}{\tau}$ ways to choose τ units of processing from those required on M_i. If we focus on a particular one of these τ units and a specific time t, then the probability that it is scheduled at time t is at most $1/\Pi_{\max}$, since we selected a delay uniformly at random from among $\Pi_{\max}$ possibilities. If ail τ units are from different jobs then the probability that they are all scheduled at time t is at most $(1/\Pi_{\max})^{\tau}$ since the delays are chosen independently. Otherwise, the probability that all τ are scheduled at time t is 0, since two units from the same job can never be assigned to the same time. Therefore

$$\begin{aligned} p &\leq \binom{\Pi_{\max}}{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau} \\ &\leq \left(\frac{e\Pi_{\max}}{\tau}\right)^{\tau}\left(\frac{1}{\Pi_{\max}}\right)^{\tau} \leq \left(\frac{e}{\tau}\right)^{\tau}. \end{aligned}$$

If

$$\tau = k\frac{\log(n\mu)}{\log\log(n\mu)}$$

then $p < (n\mu)^{-(k-1)}$. We can bound the probability that *any* machine at *any* time is assigned more than

$$k\frac{\log(n\mu)}{\log\log(n\mu)}$$

jobs by $m(P_{\max} + \Pi_{\max})p < n(P_{\max} + \Pi_{\max})(n\mu)^{-(k-1)}$. Since we have assumed that $p_{\max}$ is bounded by a polynomial in n and μ, $P_{\max} + \Pi_{\max}$ is as well. If we choose k sufficiently large, then with high probability, no more than

$$k\frac{\log(n\mu)}{\log\log(n\mu)}$$

jobs are scheduled on any machine during any unit of time. □

In the special case of unit-length operations treated by Leighton, Maggs, and Rao, a schedule $\mathcal{S}$ of length L that has at most c jobs scheduled on any machine at any unit of time can trivially be "flattened" into a valid schedule of length cL by replacing one unit of $\mathcal{S}$'s time with c units of time in which we run each of the jobs that was scheduled for that time unit (see Fig. 1).

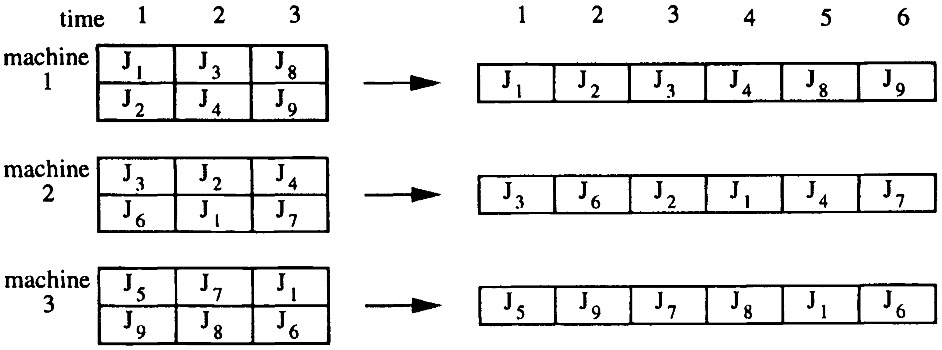


FIG. 1. *Flattening a schedule in the case with unit length operations.*

For *preemptive* job shop scheduling, where the processing of an operation may be interrupted, each unit of an operation can be treated as a unit-length operation and a schedule that has multiple operations scheduled simultaneously on a machine can easily be flattened into a valid schedule. This is not possible for *nonpreemptive* job shop scheduling, and in fact it seems to be more difficult to flatten the schedule in this case. We give an algorithm that takes a schedule of length L with at most c operations scheduled on each machine at any time and produces a schedule of length $O(cL\log p_{\max})$.

LEMMA 2.2. *Given a schedule $\mathcal{S}_0$ of length L that has at most c jobs scheduled on each machine during any unit of time, there exists a polynomial-time algorithm that produces a valid schedule of length $O(cL\log p_{\max})$.*

Proof. To begin, we round up each processing time p_{ij} to the next power of 2 and denote the corresponding rounded time by p'_{ij}; that is, $p'_{ij} = 2^{\lceil \log_2 p_{ij} \rceil}$. Let $p'_{\max} = \max_{ij} p'_{ij}$. From

$\mathcal{S}_0$, it is easy to obtain a schedule $\mathcal{S}$ that uses the modified p'_{ij} and is at most twice as long as $\mathcal{S}_0$; furthermore, an optimal schedule for the new problem is no more than twice as long as an optimal schedule for the original problem.

A *block* is an interval of a schedule with the property that each operation that begins during this interval is of length no more than that of the entire interval. (Note that this does not mean that the operation finishes within the interval.) We can divide $\mathcal{S}$ into $\lceil \frac{L}{p'_{\max}} \rceil$ consecutive blocks of size $p'_{\max}$. We will give a recursive algorithm that reschedules ("spreads") each block of size p (where p is a power of 2) into a sequence of schedule *fragments* of total length $p \log p$; the operations scheduled in a fragment of length T are all of length T, and start at the beginning of the fragment. This algorithm takes advantage of the fact that if an operation of length p is scheduled to begin in a block of size p, then that job is not scheduled on any other machine until after this block. Therefore, that operation can be scheduled to start after all of the smaller operations in the block finish.

To reschedule a block B of size $p'_{\max}$, we first construct the final fragment (which is of length $p'_{\max}$), and then construct the preceding fragments by recursive calls of the algorithm. For each operation of length $p'_{\max}$ that begins in B, reschedule that operation to start at the beginning of the final fragment, and delete it from B. Now each operation that still starts in B is of length at most $p'_{\max}/2$, so B can be subdivided into two blocks, B_1 and B_2, each of size $p'_{\max}/2$, and we can recurse on each. See Fig. 2 for an illustration of this rescheduling.

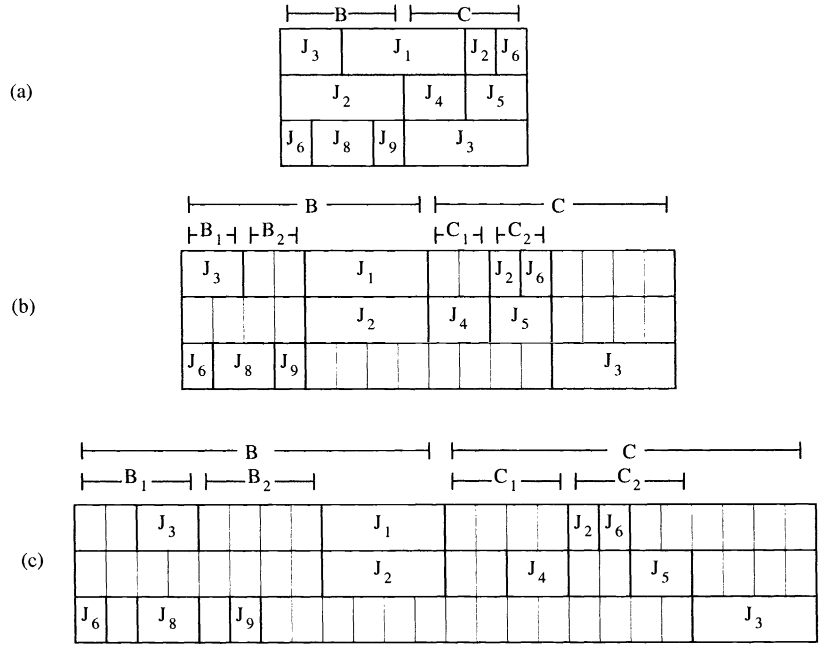


FIG. 2. (a) *The initial greedy schedule of length* 8. $p'_{\max} = 4$. (b) *The first level of spreading. All jobs of length* 4 *have been put in the final fragments. We must now recurse on* B_1 *and* B_2 *with* $p'_{\max} = 2$. (c) *The final schedule of length* $8 \log_2 8 = 24$.

The recurrence equation that describes the total length of the fragments produced from a block of size T is $f(T) = 2f(\frac{T}{2}) + T$; $f(1) = 1$. Thus $f(T) = \Theta(T \log T)$, and each block B in $\mathcal{S}$ of size $p'_{\max}$ is spread into a schedule of length $p'_{\max} \log p'_{\max}$. By spreading the schedule $\mathcal{S}$, we produce a new schedule $\mathcal{S}'$ that satisfies the following conditions:

(1) At any time in $\mathcal{S}'$, all operations scheduled are of the same length; furthermore, any two operations either start at the same time or do not overlap.

(2) If $\mathcal{S}$ has at most c jobs scheduled on one machine at any time, then this must hold for $\mathcal{S}'$ as well.

(3) $\mathcal{S}'$ schedules a job on at most one machine at any time.

(4) $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i - 1$ are completed.

Condition 1 is satisfied by each pair of operations on the same machine by the definition of spreading, and by each pair of operations on different machines because the division of time into fragments is the same on all machines. To prove condition 2, note that operations of length T that are scheduled at the same time on the same machine in the expanded schedule started in the same block of size T on that machine. Since they all must have been scheduled during the last unit of time of that block, there can be at most c of them.

To prove condition 3, note that if a job is scheduled by $\mathcal{S}'$ on two machines simultaneously that means that it must have been scheduled by $\mathcal{S}$ to start two operations of length T in the same block of length T on two different machines. This means it was scheduled by $\mathcal{S}$ on two machines during the last unit of time of that block, which violates the properties of $\mathcal{S}$.

Finally we verify condition 4 by first noting that if two operations of a job are in different blocks of size $p'_{\max}$ in $\mathcal{S}$ then they are certainly rescheduled in the correct order. Therefore it suffices to focus on the schedule produced from one block. Within a block, if an operation is rescheduled to the final fragment then it is the last operation for that job in that block. Therefore $\mathcal{S}'$ does not schedule the ith operation of job J_j until the first $i - 1$ are completed.

The schedule $\mathcal{S}'$ can easily be flattened to a schedule that obeys the constraint of one job per machine at any time, since c operations of length T that start at the same time can just be executed one after the other in total time cT. Note that since what we are doing is effectively synchronizing the entire schedule block by block, it is important when flattening the schedule to make each machine wait enough time for all machines to process all operations of that fragment length, even if some machines have no operations of that length in that fragment.

The schedule $\mathcal{S}'$ was of length $L \log p'_{\max}$; therefore the flattened schedule is of length $Lc \log p'_{\max}$. □

3. Reducing the problem. In the previous section we showed how to produce, with high probability, a schedule of length

$$O\left((\Pi_{\max} + P_{\max}) \frac{\log(n\mu)}{\log\log(n\mu)} \log p_{\max}\right),$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. Since

$$\Pi_{\max} + P_{\max} = O(\max\{\Pi_{\max}, P_{\max}\}),$$

this schedule is within a factor of

$$O\left(\frac{\log(n\mu)}{\log\log(n\mu)} \log p_{\max}\right)$$

of optimality. In this section, we will first remove the assumption that $p_{\max}$ is bounded above by a polynomial in n and μ by showing that we can reduce the general problem to that special case while sacrificing only a constant factor in the approximation. This yields an

$$O\left(\frac{\log^2(n\mu)}{\log\log(n\mu)}\right) - \text{approximation algorithm.}$$

Then we will prove a similar result that reduces this case to one in which n is bounded by a polynomial in m and μ. Combining these two results, we conclude that we can reduce the job shop scheduling problem to its special case where n and $p_{\max}$ are polynomially bounded in m and μ, while changing the performance guarantee by only a constant.

3.1. Reducing $p_{\max}$. First we will show that we can reduce the problem to one where $p_{\max}$ is bounded by a polynomial in n and μ. Let $\omega = |\mathcal{O}|$ be the total number of required operations. Note that $\omega \leq n\mu$. Round down each p_{ij} to the nearest multiple of $p_{\max}/\omega$, and denote this value by p'_{ij}. This ensures that the value p'_{ij} takes at most ω distinct values, which are all multiples of $p_{\max}/\omega$. Therefore we can treat the p'_{ij} as integers in $\{0, \dots, \omega\}$; a schedule for this problem can be trivially rescaled to a schedule $\mathcal{S}'$ for the processing times p'_{ij}. (Note that assigning $p'_{ij} = 0$ does not mean that this operation does not exist; instead, it should be viewed as an operation that takes an arbitrarily small amount of time.) Let L denote the rescaled length of $\mathcal{S}'$.

We claim that $\mathcal{S}'$ for this rounded instance can be interpreted as a schedule for the original operations of length at most $L + p_{\max}$. If we increase the processing time of O_{ij} from p'_{ij} to its original time p_{ij}, we add an amount that is at most $p_{\max}/\omega$. Since the length of a schedule is determined by a critical path through the operations and there are ω operations, we add a total amount of at most $p_{\max}$ to the length of any path; thus the new schedule is of length at most $L + p_{\max} \leq L + C^*_{\max}$. Therefore we have rounded a general instance $\mathcal{I}$ of the job shop problem to an instance $\mathcal{I}'$ for which $p_{\max} = O(n\mu)$; furthermore, a schedule for $\mathcal{I}'$ yields a schedule for $\mathcal{I}$ that is no more than $C^*_{\max}$ longer. Thus we have shown the following lemma:

LEMMA 3.1. *There exists a polynomial-time algorithm that transforms any instance of the job shop scheduling problem into one with $p_{\max} = O(n\mu)$ with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+1)C^*_{\max}$.*

3.2. Reducing the number of jobs. To reduce an arbitrary instance of job shop scheduling to one with a number of jobs that is polynomial in m and μ, we divide the jobs into big and small jobs. We say that job J_j is *big* if it has an operation of length more than $\Pi_{\max}/(2m\mu^3)$; otherwise we call the job *small*. For the instance consisting of just the short jobs, let $\Pi'_{\max}$ and $p'_{\max}$ denote the maximum machine load and operation length, respectively. Using the algorithm of [17] described in the introduction, we can, in time polynomial in the input size, produce a schedule of length $\Pi'_{\max} + 2m\mu^3 p'_{\max}$ for this instance. Since $p'_{\max}$ is at most $\Pi_{\max}/(2m\mu^3)$ and $\Pi'_{\max} \leq \Pi_{\max}$, we get a schedule that is of length no more than $2\Pi_{\max}$. Thus an algorithm that produces a schedule for the long jobs that is within a factor of k of optimal will yield a $(k+2)$-approximation algorithm. Note that there can be at most $2m^2\mu^3$ long jobs, since otherwise there would be more than $m\Pi_{\max}$ units of processing to be divided among m machines, which contradicts the definition of $\Pi_{\max}$. Thus we have shown the following lemma:

LEMMA 3.2. *There exists a polynomial-time algorithm that transforms any instance of the job shop scheduling problem into one with $O(m^2\mu^3)$ jobs with the property that a schedule for the modified instance of length $kC^*_{\max}$ can be converted in polynomial time to a schedule for the original instance of length $(k+2)C^*_{\max}$.*

From the results of the previous two sections we obtain the following theorem:

THEOREM 3.3. *There exists a polynomial-time randomized algorithm for job shop scheduling that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2(m\mu)}{\log\log(m\mu)} C^*_{\max}\right).$$

Proof. In §2 we showed how to produce a schedule of length

$$O\left((\Pi_{\max} + P_{\max})\frac{\log(n\mu)}{\log\log(n\mu)}\log p_{\max}\right)$$

under the assumption that $p_{\max}$ was bounded above by a polynomial in n and μ. From Lemmas 3.1 and 3.2 we know that we can reduce the problem to one where n and $p_{\max}$ are polynomial in m and μ, while adding only a constant to the factor of approximation. Since now $\log p_{\max} = O(\log(m\mu))$ and $\log n = O(\log(m\mu))$ our algorithm produces a schedule of length

$$O\left(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max}\right). \qquad \square$$

Note that when μ is bounded by a polynomial in m the bound only depends on m. In particular, this implies the following corollary.

COROLLARY 3.4. *There exists a polynomial-time randomized algorithm for flow shop scheduling that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2 m}{\log\log m}C^*_{\max}\right).$$

Except for the use of Sevast'yanov's algorithm, all of these techniques can be carried out in $\mathcal{RNC}$. We assign one processor to each operation. The rounding in the proof of Lemma 2.2 can be done in $\mathcal{NC}$. We set the random delays and inform each processor about the delay of its job. By summing the values of p_{ij} for all of its job's operations, each processor can calculate where its operation is scheduled with the delays and then where it is scheduled in the recursively spread out schedule. These sums can be calculated via parallel prefix operations. With simple $\mathcal{NC}$ techniques we can assign to each operation a rank among all those operations that are scheduled to start at the same time on its machine, and thus flatten the spread out schedule to a valid schedule.

COROLLARY 3.5. *There exists a $\mathcal{RNC}$ algorithm for job shop scheduling that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2(n\mu)}{\log\log(n\mu)}C^*_{\max}\right).$$

4. Applying the techniques to related problems.

4.1. A fixed number of machines. In this subsection we will show that the technique of partitioning the set of jobs by size can be applied to give a much better performance guarantee in the special case in which the number of machines and the maximum number of operations per machine are assumed to be constants. It is interesting to note that Sevast'yanov's algorithm for the job shop problem can be viewed as a $(1+m\mu^3)$-approximation algorithm, so that when m and μ are constant, this is a $O(1)$-approximation algorithm. We will give a ρ-approximation algorithm for ρ arbitrarily close to two.

To apply the idea of partitioning the jobs in this setting, call a job J_j *big* if there is an operation O_{ij} with $p_{ij} > \epsilon\Pi_{\max}/(m\mu^3)$, where ϵ is an arbitrary positive constant. Note that there are at most $m^2\mu^3/\epsilon$ big jobs, and since m, μ and ϵ are fixed, this is a constant.

Now use Sevast'yanov's algorithm to schedule all of the small jobs. The resulting schedule will be of length at most $(1+\epsilon)C^*_{\max}$. There are only a constant (albeit a huge constant) number of ways to schedule the big jobs. Therefore the best one can be selected in polynomial time and executed after the schedule of the short jobs. The additional length of this part is no more than $C^*_{\max}$.

Thus we have proven the following theorem:

THEOREM 4.1. *For any fixed value of $\epsilon > 0$, there is a polynomial-time algorithm for the special case of the job shop scheduling problem where both m and μ are fixed that produces a schedule of length $\leq (2+\epsilon)C^*_{\max}$.*

4.2. Dag shop scheduling with identical copies of machines. The fact that the quality of our approximations is based solely on the lower bounds $\Pi_{\max}$ and $P_{\max}$ makes it quite easy to extend our techniques to the more general problem of *dag shop scheduling*, in which each job is given by a specified partial order of operations. We define $\Pi_{\max}$ and $P_{\max}$ exactly the same way, and $\max\{\Pi_{\max}, P_{\max}\}$ remains a lower bound for the length of any schedule. We can reduce this dag shop scheduling problem to a job shop problem by selecting for each job an arbitrary total order that is consistent with its partial order. The maximum job length and maximum machine load for the job shop instance constructed are equal to the analogous values for the original dag shop instance. Therefore, a schedule of length $\rho \cdot (\Pi_{\max} + P_{\max})$ for this job shop instance is a schedule for the original dag shop scheduling instance of length $O(\rho C^*_{\max})$.

A further generalization to which our techniques apply is when, rather than m different machines, we have m' types of machines, and for each type we have a specified number of identical machines of that type. Instead of requiring an operation to run on a particular machine, an operation now has to run on only one of these identical copies. $P_{\max}$ remains a lower bound on the length of any schedule for this problem. $\Pi_{\max}$, which was a lower bound for the job shop problem, must be replaced, since we do not have a specific assignment of operations to machines and the sum of the processing times of all operations assigned to a type is *not* a lower bound. Let S_i, $i = 1, \ldots, m'$, denote the sets of identical machines, and let $\Pi(S_i)$ be the sum of the lengths of the operations that run on S_i. Our strategy is to reduce this to a job shop problem by assigning operations to specific machines in such a way that the maximum machine load is within a constant factor of the fundamental lower bounds for this problem. For each set of machines S_i, $i = 1, \ldots, m'$, the average load on that set of machines is clearly a lower bound on the maximum machine load of machines within that set; thus

$$\Pi_{\text{avg}} = \max_{S_i} \frac{\Pi(S_i)}{|S_i|}$$

is a lower bound on the maximum machine load. Furthermore, we cannot split operations, so $p_{\max}$ is also a lower bound. We will now describe how to assign operations to machines so that the maximum machine load of the resulting job shop scheduling problem is at most $2\Pi_{\text{avg}} + p_{\max}$. A schedule for the resulting job shop problem of length $\rho \cdot (\Pi_{\max} + P_{\max})$ yields a solution for the more general problem of length $O(\rho \cdot (\Pi_{\text{avg}} + P_{\max}))$. Sevast'yanov [17] used a somewhat more complicated reduction to handle a slightly more general setting.

For each operation O_{ij} to be processed by a machine in S_k, if $p_{ij} \geq \Pi(S_k)/|S_k|$, assign O_{ij} to one machine in S_k. There are certainly enough machines in S_k to assign these operations so that each machine is assigned at most one of them; this contributes at most $p_{\max}$ to the maximum machine load. Those operations not yet assigned are each of length at most $\Pi(S_k)/|S_k|$ and have total length $\leq \Pi(S_k)$. Therefore, these can be assigned easily to the remaining machines so that less than $2\Pi(S_k)/S_k$ processing units are assigned to each machine. Combining these two bounds, we get an upper bound on the maximum machine load of $2\Pi_{\text{avg}} + p_{\max}$, which is within a constant factor of the lower bound of $\max\{\Pi_{\text{avg}}, p_{\max}\}$.

THEOREM 4.2. *There exists a polynomial-time randomized algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2(m\mu)}{\log\log(m\mu)}C^*_{\max}\right).$$

COROLLARY 4.3. *There exists an $\mathcal{RNC}$ algorithm for dag shop scheduling with identical copies of machines that, with high probability, yields a schedule that is of length*

$$O\left(\frac{\log^2(n\mu)}{\log\log(n\mu)}C^*_{\max}\right).$$

4.3. Unrelated parallel machines with chain precedence constraints. A further generalization of the job shop problem is the problem of scheduling jobs on unrelated parallel machines subject to chain precedence constraints, which is denoted $R|chain|C_{\max}$ in the notation of [7]. In this problem, we are given a set of jobs $\mathcal{J} = \{J_1, \dots, J_n\}$, each of which is to be processed by exactly one of a set of machines $\mathcal{M} = \{M_1, \dots, M_m\}$; if J_j is processed by M_i, it takes p_{ij} time units. We are also given a partial order $\prec$ that specifies job precedence constraints: if $J_j \prec J_k$ then J_k may not start processing until J_j has completed processing. The precedence constraints are restricted to be a disjoint union of chains; that is, we are given a set $\mathcal{C} = \{C_1, \dots, C_k\}$, where each chain $C_l \in \mathcal{C}$ consists of some μ_l jobs, given with a linear ordering on these jobs; each job is in exactly one chain (where unconstrained jobs are viewed as chains of length one). The objective is to construct a schedule consistent with the precedence constraints so as to minimize the maximum job completion time $C_{\max}$.

We will view the problem of finding a feasible solution for this problem as having two phases: finding an assignment of jobs to machines, and then constructing a schedule consistent with that assignment. It is easy to see that the second phase is a job shop scheduling problem: each operation of the job shop problem corresponds to a job of the parallel machines problem, and each job of the job shop problem corresponds to a chain of the parallel machines problem. We will construct an assignment such that the resulting job shop instance has $\Pi_{\max} \leq 4L$ and $P_{\max} \leq 2L$, where L is a lower bound for the original instance of $R|chain|C_{\max}$. By applying our approximation algorithms for the job shop problem to this instance, we obtain approximation algorithms for $R|chain|C_{\max}$ with an identical performance guarantee (up to a constant factor).

To find a suitable assignment, we give an algorithm that takes a given threshold value d, and either proves that any feasible schedule for the instance of $R|chain|C_{\max}$ has $C_{\max} > d$, or else constructs an assignment such that the resulting job shop instance has $\Pi_{\max} \leq 4d$ and $P_{\max} \leq 2d$. By performing a bisection search with initial lower bound 0 and upper bound $\sum_j \max_i p_{ij}$, we obtain a polynomial-time algorithm that finds a value L such that no schedule of length $L - 1$ exists (and hence L is a valid lower bound), and we have an assignment such that the resulting job shop instance has $\Pi_{\max} \leq 4L$ and $P_{\max} \leq 2L$.

The algorithm that tests a given threshold is based on results of Lenstra, Shmoys, and Tardos [9] and Lin and Vitter [10]. It works by first constructing an integer program which must be feasible if there is a schedule of length d, and then checks if its linear relaxation is feasible. If the linear program is infeasible, then no schedule of length d exists, and if the linear program is feasible, then the fractional solution can be rounded to yield an integer assignment with the claimed properties.

Observe that if there is a schedule with $C_{\max} \leq d$, then the following linear program has a feasible solution:

$$\begin{aligned}
\sum_{j=1}^{n} p_{ij}x_{ij} &\leq d, & i &= 1, \ldots, m, \\
\sum_{i=1}^{m} \sum_{j \in C_l} p_{ij}x_{ij} &\leq d, & l &= 1, \ldots, k, \\
\sum_{i=1}^{m} x_{ij} &= 1, & i &= 1, \ldots, m, \\
x_{ij} &\geq 0, & i &= 1, \ldots, m, \qquad j = 1, \ldots, n.
\end{aligned}$$

Let x_{ij} denote such a feasible solution; we can view this solution as a fractional assignment where an x_{ij} fraction of job J_j is assigned to M_i. Let p_j denote the total time spent processing J_j in this fractional assignment; that is,

$$p_j = \sum_{i=1}^{m} p_{ij}x_{ij}, \qquad j = 1, \ldots, n.$$

In this fractional assignment, it is possible for J_j to have a small fraction assigned to a machine M_i for which its p_{ij} value is substantially more than p_j. To make sure that this does not occur, we first apply the filtering technique of Lin and Vitter [10], and round x_{ij} to $\bar{x}_{ij}$ by setting

$$\bar{x}_{ij} = \begin{cases} x_{ij}, & \text{if } p_{ij} \leq 2p_j, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Of course, for each job J_j, only a fraction $t_j = \sum_{i=1}^{m} \bar{x}_{ij}$ of it is now assigned. We renormalize this fractional assignment by forming $\tilde{x}_{ij} = \bar{x}_{ij}/t_j$. Observe that $t_j \geq 1/2$, $j = 1, \ldots, n$, since less than half of each job J_j can be assigned to any machine M_i for which $p_{ij} > 2p_j$. Therefore $\tilde{x}$ satisfies

$$\begin{aligned}
&\sum_{j=1}^{n} p_{ij}\tilde{x}_{ij} \leq 2d, && i = 1, \ldots, m, \\
&\sum_{i=1}^{m} \tilde{x}_{ij} = 1, && i = 1, \ldots, m, \\
&\tilde{x}_{ij} = 0, \quad \text{if } p_{ij} > 2p_j, && i = 1, \ldots, m, \qquad j = 1, \ldots, n, \\
&\tilde{x}_{ij} \geq 0, && i = 1, \ldots, m, \qquad j = 1, \ldots, n.
\end{aligned}$$

A result of Lenstra, Shmoys, and Tardos [9] states that, in polynomial-time, any extreme point of such a linear program can be rounded to an integer solution x^* such that

$$\begin{aligned}
&\sum_{j=1}^{n} p_{ij}x^*_{ij} \leq 2d + 2\max_j\{p_j\}, && i = 1, \ldots, m, \\
&\sum_{i=1}^{m} x^*_{ij} = 1, && i = 1, \ldots, m, \\
&x^*_{ij} = 0, \quad \text{if } p_{ij} > 2p_j, && i = 1, \ldots, m, \qquad j = 1, \ldots, n, \\
&x^*_{ij} \geq 0, && i = 1, \ldots, m, \qquad j = 1, \ldots, n.
\end{aligned}$$

Furthermore, since $\sum_{j \in C_l} p_j \leq d, l = 1, \ldots, k$, we have that

$$\sum_{j \in C_l} \sum_{i=1}^{m} p_{ij} x_{ij}^* \leq \sum_{j \in C_l} 2p_j \leq 2d.$$

Consequently, if for each job J_j we assign the corresponding operation to the machine M_i for which $x_{ij}^* = 1$, then we obtain a job shop instance for which $P_{\max} \leq 2d$, and $\Pi_{\max} \leq 2d + 2 \max_j \{p_j\} \leq 4d$, as claimed.

THEOREM 4.4. *There exists a polynomial-time randomized algorithm for $R|chain|C_{\max}^*$ that, with high probability, yields a schedule that is of length at most*

$$O\left(\frac{\log^2 n}{\log \log n} C_{\max}^*\right).$$

5. A deterministic approximation algorithm. In this section, we "derandomize" the results of the previous sections: we give a deterministic polynomial-time algorithm that finds a schedule of length $O(\log^2(m\mu)C_{\max}^*)$. Of all the components of the algorithm of Theorem 3.3, the only step that is not already deterministic is the step that uniformly chooses a random initial delay for each job with the resulting property that, with high probability, no machine is assigned too many jobs at any time. In particular, the reduction to the special case in which n and $p_{\max}$ are bounded by a polynomial in m and μ is entirely deterministic, and so we can focus on that case alone.

We will give an algorithm that deterministically assigns delays to each job so as to produce a schedule in which each machine has $O(\log(m\mu))$ jobs running at any one time. We then apply Lemma 2.2 to produce a schedule of length $O(\log^2(m\mu)C_{\max}^*)$. Note that the $O(\log(m\mu))$ jobs per machine is not as good as the probabilistic bound of

$$O\left(\frac{\log(m\mu)}{\log \log(m\mu)}\right).$$

Recently, Schmidt, Siegel, and Srinivasan [15] have given a different derandomizing strategy for this problem that yields delays that match the performance of the randomized algorithm.

Our approach to the problem of selecting good delays is to frame it as a *vector selection* problem and then apply techniques developed by Raghavan and Thompson [13], [14] and Raghavan [12] that find constant factor approximations to certain "packing" integer programs. The approach is to formulate the problem as a {0, 1}–integer program, solve the linear programming relaxation, and then randomly round the solution to an integer solution. For certain types of problems this yields provably good approximations with high probability [13], [14]. Furthermore, for many of the problems for which there are approximations with high probability, the algorithm can be derandomized. Raghavan [12] has shown how to do this by essentially setting the random bits one at a time.

We now state the problem formally:

PROBLEM 5.1. *Deterministically assign a delay to each job in the range* $[0, \Pi_{\max}/\log(m\mu)]$ *so as to produce a schedule with* $O(\log(m\mu))$ *jobs on any machine at any time.*

LEMMA 5.2. *Problem* 5.1 *can be solved in deterministic polynomial time.*

Proof. Since we introduce delays in the range $[0, \Pi_{\max}/\log(m\mu)]$, the resulting schedule has length $\ell = P_{\max} + \Pi_{\max}/\log(m\mu)$. We can represent the processing of a job J_j with a given initial delay d by $\{0, 1\}$–vector of length $\ell \cdot m$, where each component corresponds to a particular machine at a particular time. The position corresponding to machine M_i and time

t is 1 if M_i is processing job J_j at time t, and 0 otherwise. For each job J_j and each possible delay d, there is a vector $V_{j,d}$ that corresponds to assigning delay d to J_j.

Let λ_j be the set of vectors $\{V_{j,1}, \dots, V_{j,d_{\max}}\}$, where $d_{\max} = \Pi_{\max}/\log(m\mu)$, and let $V_{j,k}(i)$ be the ith component of $V_{j,k}$. Given the set $\Lambda = \{\lambda_1, \dots, \lambda_n\}$ of sets of vectors, our problem can be stated as the problem of choosing one vector V_j^* from each λ_j such that

$$\left\| \sum_{j=1}^{n} V_j^* \right\|_\infty = O(\log(m\mu));$$

that is, at any time on any machine, the number of jobs using that machine is $O(\log(m\mu))$.

As in [12], we can reformulate this as a $\{0, 1\}$–integer program. Let $x_{j,k}$ be the indicator variable used to indicate whether $V_{j,k}$ is selected from λ_j. Consider the integer program (IP) that assigns $\{0, 1\}$ values to the variables $x_{j,k}$ to minimize W subject to the constraints

$$\sum_{k=1}^{d_{\max}} x_{j,k} = 1, \qquad j = 1, \dots, n,$$

$$\sum_{j=1}^{n} \sum_{k=1}^{d_{\max}} x_{j,k} V_{j,k}(i) \le W, \qquad i = 1, \dots, \ell \cdot m.$$

Let W_{OPT} be the optimum value of W, which is the maximum number of jobs that ever use a machine at any time. We already know, by Lemma 2.1, that $W_{\mathrm{OPT}} = O(\log(m\mu))$, and so an optimal solution to this integer program would solve Problem 5.1. However, the problem is $\mathcal{NP}$-hard. Instead, we rely on the following theorem, which is immediate from the results in [12] and [14].

THEOREM 5.3 [12], [14]. *A feasible solution to (IP) with $W = O(W_{\mathrm{OPT}} + \log(m\mu))$ can be found in polynomial time.*

We then apply Lemma 2.2 to obtain the following result:

THEOREM 5.4. *There exists a deterministic polynomial-time algorithm that finds a schedule of length $O(\log^2(m\mu)\, C^*_{\max})$.* □

6. Conclusions and open problems. We have given the first polynomial-time polylog-approximation algorithms for minimizing the maximum completion time for the problems of job shop scheduling, flow shop scheduling, dag shop scheduling, and several other generalizations.

One particularly simple special case of dag shop scheduling can be obtained if the partial order for each job is empty; in other words, each job consists of a number of operations that may be performed in any order. This is called the *open shop* problem, and it is traditional in the scheduling literature to focus on the case when each job is processed on each machine at most once (since operations on the same machine can be coalesced).

A consequence of our results is the following observation about the structure of shop scheduling problems. Assume we have a set of jobs that need to run on a set of machines. We know that any schedule for the associated open shop problem must be of length $\Omega(\Pi_{\max} + P_{\max})$. Furthermore, we know that no matter what type of partial ordering we impose on the operations of each job we can produce a schedule of length

$$O\left((\Pi_{\max} + P_{\max}) \frac{\log^2 m}{\log \log m} \right).$$

Hence for any instance of the open shop problem, we can impose an arbitrary partial order on the operations of each job and increase the length of the optimal schedule by a factor of

$$O\left(\frac{\log^2 m}{\log\log m}\right).$$

On the other hand, there does exist a schedule of length $O(P_{\max}+\Pi_{\max})$ for the open shop problem. Consider the simple greedy algorithm that, whenever a machine is idle, assigns to it any job that has not yet been processed on that machine and is not currently being processed on another machine. Anná Racsmány [2] has observed that the greedy algorithm delivers a schedule of length at most $\Pi_{\max} + (m - 1)p_{\max}$. We can adapt her proof to show that, in fact, the greedy algorithm delivers a schedule that is of length less than $\Pi_{\max} + P_{\max} \le 2C^*_{\max}$. Consider the machine M_k that finishes last in the greedy schedule; this machine is active sometimes, idle sometimes, and finishes by completing some job J_j. Since the schedule is greedy, whenever M_k is idle, J_j is being processed by some other machine, and so the idle time is at most $\sum_{M_i \ne M_k} p_{ij} < P_{\max}$. Thus machine M_k is processing for at most $\Pi_{\max}$ units of time and is idle for less than $P_{\max}$ units of time; hence $C_{\max} < \Pi_{\max} + P_{\max}$. Fiala [5] has also shown that if $\Pi_{\max} \ge (16m\log m + 21m)p_{\max}$, then $C^*_{\max}$ is just $\Pi_{\max}$, and there is a polynomial-time algorithm to find an optimal schedule.

We have seen that in two interesting special cases, job shop scheduling with unit-length operations and open shop scheduling, there is a schedule of length $O(\Pi_{\max} + P_{\max})$, and so the major open question left unresolved by this paper is:

- Does there exist an $O(\Pi_{\max}+P_{\max})$ schedule for the general job or flow shop scheduling problem? If so, when can it be found in polynomial time?

Beyond this, there are a number of interesting questions raised by this work, including

- Do there exist parallel algorithms that achieve the approximations of our sequential algorithms? For the general job shop problem this seems hard, since we rely heavily on the algorithm of Sevast'yanov. For open shop scheduling, however, a simple sequential algorithm achieves a factor of two, whereas the best $\mathcal{NC}$ algorithm that we have achieves only an $O(\log n)$-approximation. As a consequence of the results above, all one would need to do is to produce any greedy schedule.
- Are there simple variants of the greedy algorithm for open shop scheduling that achieve better performance guarantees? For instance, how good is the algorithm that always selects the job with the maximum total (remaining) processing time? Williamson et al. [20] have shown that the existence of a ρ-approximation algorithm with $\rho < 5/4$ would imply that $\mathcal{P} = \mathcal{NP}$. It remains an interesting open problem to close this gap.
- Our algorithms, while polynomial-time algorithms, are inefficient. Are there significantly more efficient algorithms which have the same performance guarantees? Stein [19] has given an algorithm that directly finds a good approximate solution to the integer program (IP) by using the framework of Plotkin, Shmoys, and Tardos [11]. This yields an implementation of our algorithm that runs in $O(n^2m^2\mu^2 + n^3\mu^2\log(m\mu)(\mu+\log(m\mu)))$ time. Although this represents a dramatic improvement over the previously known bound, it remains an interesting question to give substantially more efficient algorithms.

Acknowledgments. We are grateful to David Williamson for working with us in the early stages of this research, and to Imre Bárány, Tom Leighton, Bruce Maggs, and Yishay Mansour for many helpful discussions. We thank Jim Orlin for the observation about the gap between open shop and dag shop scheduling.

REFERENCES

[1] D. Applegate and W. Cook, *A computational study of the job-shop scheduling problem*, ORSA Journal of Computing, 3 (1991), pp. 149–156.

[2] I. Bárány and T. Fiala, *Többgépes ütemezési problémák közel optimális megoldása*, Szigma–Mat.–Közgazdasági Folyóirat, 15 (1982), pp. 177–191.

[3] I. Belov and Y. N. Stolin, *An algorithm in a single path operations scheduling problem*, in Mathematical Economics and Functional Analysis [In Russian], Nauka, Moscow, 1974, pp. 248–257.

[4] T. Fiala, *Közelítő algorithmus a három gép problémára*, Alkalmazott Matematikai Lapok, 3 (1977), pp. 389–398.

[5] ———, *An algorithm for the open-shop problem*, Math. Oper. Res., 8 (1983), pp. 100–109.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[7] E. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *Sequencing and scheduling: Algorithms and Complexity*, in Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory, S. C. Graves, A. H. G. Rinnooy Kan, and P. H. Zipkin, eds., North Holland, Amsterdam, 1993, pp. 445–522.

[8] T. Leighton, B. Maggs, and S. Rao, *Universal packet routing algorithms*, in Proceedings of the 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 256–269.

[9] J. K. Lenstra, D. B. Shmoys, and E. Tardos, *Approximation algorithms for scheduling unrelated parallel machines*, Math. Programming, 46 (1990), pp. 259–271.

[10] J.-H. Lin and J. S. Vitter, *ϵ-approximation with minimum packing constraint violation*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 771–782.

[11] S. Plotkin, D. B. Shmoys, and E. Tardos, *Fast approximation algorithms for fractional packing and covering problems*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, D.C., 1991, pp. 495–504.

[12] P. Raghavan, *Probabilistic construction of deterministic algorithms: approximating packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.

[13] P. Raghavan and C. D. Thompson, *Provably good routing in graphs: regular arrays*, in Proceedings of the 17th Annual ACM Symposium on Theory of Computing, New York 1985, pp. 79–87.

[14] ———, *Randomized rounding: a technique for provably good algorithms and algorithmic proofs*, Combinatorica, 7 (1987), pp. 365 - 374.

[15] J. Schmidt, A. Siegel, and A. Srinivasan, *Chernoff-Hoeffding bounds for applications with limited independence*, in Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA 1993, pp. 331–340.

[16] S. Sevast'yanov, *Efficient construction of schedules close to optimal for the cases of arbitrary and alternative routes of parts*, Soviet Math. Dokl., 29 (1984), pp. 447–450.

[17] ———, *Bounding algorithm for the routing problem with arbitrary paths and alternative servers*, Kibernetika, 22 (1986), pp. 74–79. Translation in Cybernetics 22, pp. 773–780.

[18] ———, *On an asymptotic approach to some problems in scheduling theory*, in Abstracts of papers at 3rd All-Union Conf. of Problems of Theoretical Cybernetics [in Russian], Inst. Mat. Sibirsk. Otdel. Akad. Nauk SSSR, Novosibirsk, 1974, pp. 67–69.

[19] C. Stein, *Approximation algorithms for multicommodity flow and shop scheduling problems*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, Aug. 1992. Also appears as MIT/LCS/TR-550.

[20] D. P. Williamson, L. A. Hall, J. A. Hoogeveen, C. A. J. Hurkens, J. K. Lenstra, and D. B. Shmoys, *Short shop schedules*, unpublished manuscript, 1993.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 633–651, June 1994

011

RANDOMIZED ALGORITHMS FOR BINARY SEARCH AND LOAD BALANCING ON FIXED CONNECTION NETWORKS WITH GEOMETRIC APPLICATIONS*

JOHN H. REIF† AND SANDEEP SEN‡

Abstract. There are now a number of fundamental problems in computational geometry that have optimal algorithms on PRAM models. This paper presents randomized parallel algorithms that execute on an n-processor butterfly interconnection network in $O(\log n)$ time for the following problems of input size n: trapezoidal decomposition, visibility, triangulation, and two-dimensional convex hull. These algorithms involve tackling some of the very basic problems, like binary search and load balancing, that are taken for granted in PRAM models. Apart from a two-dimensional convex hull algorithm, these are the first nontrivial geometric algorithms that attain this performance on fixed connection networks. These techniques use a number of ideas from *Flashsort* that have to be modified to handle more difficult situations; it seems likely that they will have wider applications.

Key words. parallel algorithms, randomization, computational geometry, butterfly network

AMS subject classifications. 68E05, 68C05, 68C25

1. Introduction.

1.1. Motivation and overview.

In the past decade, we have witnessed a systematic growth in state-of-the-art parallelizing algorithms in the Parallel Random Access Machine (PRAM) environment. As a result, a number of basic problems have been recognized and many sophisticated techniques have been developed—these can be viewed as a "tool-kit" for tackling increasingly complex problems. There is a general consensus that PRAM models are appropriate for the algorithm designer but that these algorithms have to be implemented on fixed-connection networks to be of any practical significance. Using well-known general-purpose emulation schemes, all these algorithms can be implemented to run on a butterfly (or a hypercube) network with a $O(\log n)$ multiplicative factor degradation in time complexity. The crucial question is: Can PRAM algorithms be extended to fixed-connection networks without this logarithmic penalty in running time?

In a top–down approach to algorithm design, complicated algorithms are built on top of less complex procedures. The answer to the above question would depend on how far down in this hierarchy one can go without running into problems that cannot be mapped optimally on the fixed-connection network. Moreover, this would also depend on the nature of the algorithm itself. One of the most basic problems in this hierarchy is sorting. For example, Reischuk's [19] $O(\log n)$ time, n processor randomized PRAM sorting algorithm was successfully extended to networks by Reif and Valiant [18] to run in $O(\log n)$ time by using additional new sampling techniques for problem-size control. In contrast, Cole's deterministic $O(\log n)$ time parallel mergesort algorithm seems prohibitively difficult to implement (without a logarithmic slowdown) on the networks because of its liberal use of pointers. Consequently a number of algorithms that use this approach on PRAM models would be at least as difficult as being adapted to network models.

*Received by the editors July 9, 1990; accepted for publication (in revised form) March 16, 1993. This research was supported in part by Air Force Office of Scientific Research contract AFSOR-87-0386, Office of Naval Research contract N00014-87-K-0310, National Science Foundation contract CCR-8696134, Defense Advanced Research Projects Agency/Army Research Office contract DAAL03-88-K-0185, and Defense Advanced Research Projects Agency/ISTO contract N00014-88-K-0458. A preliminary version of this paper appeared in the Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures, Greece, 1990.

†Department of Computer Science, Duke University, Durham, North Carolina 27706.

‡Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India.

Although the eventual goal of this paper is to present efficient algorithms on interconnection networks, we would like the reader to view this in the more general context of mapping certain kinds of PRAM algorithms on fixed-connection networks and the difficulties associated therewith. We encountered several problems that appear to be very basic for this line of research, and we believe that these will have much wider applications.

Remark. In this paper, the term fixed-connection network has been used to allude to networks that have $O(\log n)$ diameter for n-node networks. There already exists a large body of literature for geometric algorithms on grid-like networks where the diameter is a bottleneck for achieving the kind of time complexity for which we are aiming.

One of the underlying problems is doing binary search optimally in a model that does not allow concurrent reads. A common scenario is the following: we are given a tree whose leaves represent intervals and n keys for which we have to determine the interval in which each key lies. If the depth of this tree is d, then it is trivial to do this sequentially in $O(nd)$ time. In case of PRAM models that allow concurrent reads, the problem is again quite simple. With n processors, we can simultaneously search for all keys in $O(d)$ parallel time, thus resulting in an optimal speed-up. The main difficulty associated with this problem stems from the possibility that the keys may be unevenly distributed among the intervals. In this paper we have described a randomized algorithm to do this in an $EREW$ PRAM in $O(\log n)$ time and refined it further to run in the same asymptotic time on an n-node butterfly network.

An additional problem is that of allocation of subproblems to subnetworks for recursive calls. Unlike $PRAM$ models, the network topology imposes severe constraints on processor allocation — not only should the number of processors match with the subproblem sizes but they should also be interconnected in a certain manner. Our solution to this problem could be applied to a more general situation than the applications described in this paper. Some of the ideas are similar to *Flashsort* where one does *splitter-directed routing* to route the keys to the appropriate subnetworks. However, unlike *Flashsort*, we may be confronted with situations where we have to allocate resources dynamically, as the subproblems could have varying sizes. One of our results is that we have near-optimal solutions to the above problems that should have applications to a wide class of algorithms. These basic procedures serve as a crucial link between the PRAM algorithms and interconnection networks.

1.2. Geometry on fixed-connection networks and main results. Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. There have been two fundamentally distinct approaches to this area of research, namely, the deterministic methods and algorithms that use random sampling. Some of the earliest work in this area was by Chow [4], who developed algorithms for a number of fundamental problems that were deterministic and executed in interconnection networks with polylogarithmic running time. A more general approach for deterministic PRAM algorithms was pioneered by Aggarwal et al. [1], who developed some new techniques for designing efficient parallel algorithms for fundamental geometric problems. However, a majority of the algorithms were not optimal in $P \cdot T$ bounds. A number of the most efficient deterministic PRAM algorithms are from Atallah, Cole, and Goodrich [2], who extended the techniques used by Cole [8] for his parallel mergesort algorithm. Their technique is called *Cascaded merging* and has been subsequently used (independently by Chandran [3]) for a number of other problems. Note that most of the geometric problems in the context of research in parallel algorithms have sequential time complexity of $\Omega(n \log n)$ and a typical performance that one aims for is $O(\log n)$ parallel time using an optimal number of processors.

In an independent development, Reif and Sen [17] were also able to derive $O(\log n)$ time-optimal algorithms for point-location and trapezoidal decomposition, which were randomized. Later they extended their methods to give optimal algorithms for three-dimensional convex hulls (and hence two-dimensional Voronoi diagrams) on the *CREW* PRAM model. At the core of their algorithms were random sampling techniques that had also been introduced by Clarkson [5], [6], [7] and Haussler and Welzl [10]. In addition, a new resampling technique called *Polling* was used successfully to derive the parallel algorithms. In essence, it is an efficient resampling procedure that enables us to choose a sample that satisfies certain properties with high probability from a set of samples for which only expected behavior is known. While no deterministic algorithms have been developed for some of the above problems that attain optimal bounds, we feel that the real impact of randomized techniques will be in the domain of parallel algorithms on fixed-connection networks.

In spite of interesting developments in the PRAM world, the state-of-art of geometric algorithms in the case of small-diameter fixed-connection networks is lagging far behind. Presently, the only known $O(\log n)$ time algorithm for the network model is a two-dimensional convex hull algorithm due to Miller and Stout [13]. Consequently, the *Cascaded-merge technique* of Atallah, Cole, and Goodrich [2], in spite of its elegance on the PRAM models, appear to be of little use in a fixed-connection model. The only obvious way to implement pointer updates takes $O(\log n)$ time per step of the PRAM algorithm, which results, in an $\tilde{O}(\log^2 n)$ time algorithm.

The randomized algorithms seem to be more amenable to mapping on fixed-connection networks, although they are far from straightforward. We derive an optimal $O(\log n)$ time randomized algorithm for constructing the trapezoidal decomposition. Using this, we can triangulate a simple polygon in $O(\log n)$ time.

1.3. Model of computation and notation. Throughout this paper we will be using the butterfly interconnection model where the processors operate in a synchronous fashion and have bounded buffer size. These assumptions are consistent with some of the existing machines such as the BBN Butterfly and Connection Machine. During every step, each processor is allowed to perform a real-arithmetic operation consistent with standard models used for sequential geometric algorithms. Moreover, each processor has access to a random-number generator that returns in unit time a truly random number of $O(\log n)$ bits. One of the primary reasons for choosing the butterfly network is because of its "recursive" nature. A butterfly network of order k (which will be referred to as BF_k) has k levels of 2^k nodes each, i.e., it has $k2^k$ nodes. Each node has an address (w, t) where $w \in \{0, 1\}^k$ and $0 \leq t < k$ (see Fig. 1). The first component of this address will be referred to as *row* and the second component as *rank*. The significance of this network is that there are numerous "copies" of BF_l in BF_k for $l < k$. We shall make crucial use of the following fact.

Fact 1. For any w_1, w_2 such that $|w_1| + |w_2| = k - l$, the subgraph of BF_k spanned by the nodes $\{(w_1 w w_2, i) | w \in \{0, 1\}^l\}$ and $|w_1| \leq i \leq |w_1| + l$ is isomorphic to BF_l.

Here $|w|$ is used to denote the length of the (binary) string w. Moreover, we may need to emulate a larger butterfly in a work-preserving fashion. The following simple result can be used.

Fact 2. A BF_k can emulate a BF_{k+c} within $O(3^c)$ slowdown where c is a positive integer.

This is similar to Brent's slowdown lemma except that we must construct a mapping of the processors of BF_{k+c} to BF_k respecting the interconnection topology. In this case it is very straightforward. Assume that $c = 1$ and map the processors with addresses (xw, i) where $x \in \{0, 1\}$ and $|w| = k$ and $i \leq k$ to (w, i). One can verify that the processors of BF_{k+1} that were neighbors are neighbors in the smaller network. Each processor in the smaller network has to do at most twice the amount of work and requires twice the amount of local memory. Only the processors of rank k must do the extra work of emulating the processors of rank

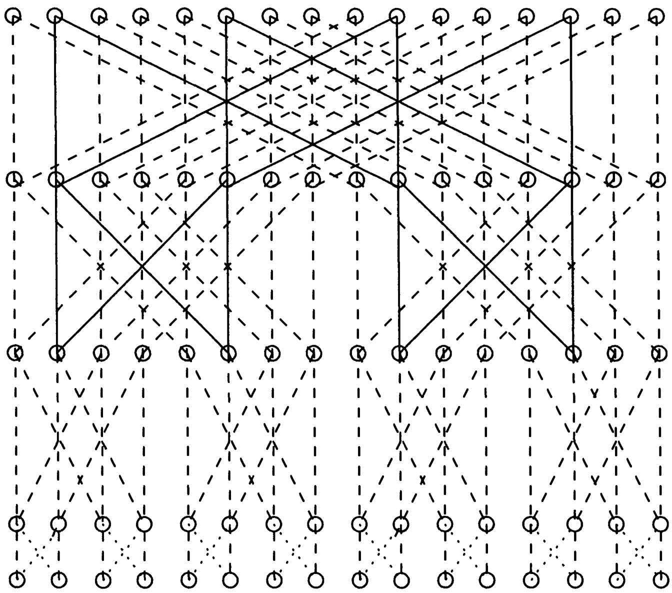


FIG. 1. *A butterfly network of order* 4. *The solid lines illustrate a subnetwork isomorphic to* BF_2.

$k+1$. This scheme can be extended directly to yield the claimed bound. For a fixed c, this implies that there is a constant factor slow-down. In this paper this scheme will often be used with the value of c being 1 or 2.

The term *very high likelihood* (*probability*) is used in this paper to denote probability $> 1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size. Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we shall use $\tilde{O}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}(f(n))$ if there is a constant c such that the resource used by the algorithm is no more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$ for an input of size n. (An equivalent definition will bound the resource by $\alpha \cdot f(n)$ with probability greater than 1 - $n^{-c\alpha}$, and in the rest of the paper they will be used in an interchangeable manner.)

The rest of the paper is organized as follows. In §2, we briefly review two important subroutines in a fixed-connection network, namely, sorting and routing. In §3, we describe an algorithm for conducting a binary search. This is used in §4 to develop fast methods for searching in arrangements. We give a brief description of the algorithm for trapezoidal decomposition. We focus mainly on those portions for which we need different methods from the PRAM algorithms — the reader is referred to a previous paper (Reif and Sen [17]) for a more detailed description of the algorithms that were developed for the $CREW$ PRAM model.

2. Overview of sorting and routing on fixed-connection networks. Our algorithms use sorting and routing extensively at various stages, and a brief review of these routines will help us in understanding the latter algorithms that use them as building blocks. The problem of *packet routing* involves routing a message from processor i to $\Pi(i)$ for all i where Π is a permutation function. There has been a long and rich history of routing algorithms for fixed connection networks (see [22], [21], [15], [11]), and these can be summarized as follows.

LEMMA 2.1. *There exists an algorithm for permutation routing on an n-node butterfly network that executes in* $\tilde{O}(\log n)$ *steps and uses only constant size queues to achieve this running time.*

A more general result has been proved by Maggs et al. [21] for *layered* networks. A *layered* network is one whose nodes can be assigned layer numbers and in which each edge connects a layer i node to a layer $i+1$ node (butterfly is an example of such a network). Let d denote the maximum distance traveled by any packet and c the largest number of packets that must traverse a single edge (this parameter c is also called the congestion). These parameters are fixed for a given selection of paths by all the packets to be routed. Then there exists a scheme for scheduling the movements of the packets such that with high probability the routing can be completed in $O(c + d + \log n)$ steps where n is the size of the network and $O(n)$ packets are being routed.

Remark. Given the above result and the fact that d is $O(\log n)$ for most path selection strategies, especially in a butterfly network, it remains to bound the value of c to get a bound on the routing time. For packets being routed to a random location, c can be bounded by $O(\log n)$ with high probability.

The first optimal $\tilde{O}(\log n)$ time-sorting algorithm, called *Flashsort*, for the butterfly network was by Reif and Valiant [18]. It was based on a PRAM sorting algorithm by Reischuk [19] but required several additional techniques because of the constraints imposed by the network connectivity. A slightly simplified version can be presented as follows.

1. Select n^{ϵ_0}-($\epsilon_0 < 1/2$) sized random subset from the given set of n keys.

2. Sort these, using a simple method, like doing all the pairwise comparisons and ranking them.

3. Use these keys to set up a binary tree such that the leaves of the tree correspond to the intervals defined by a pair of consecutive splitter keys. Over-sampling techniques are used to ensure that these intervals partition the remaining keys into roughly equal-sized subsets. This eliminates the need for dynamic load balancing in the special case of sorting. The keys are assumed to be in random locations initially. For each subset a subnetwork of appropriate size is set aside and the keys that belong to this subset are routed to this part of the network. This is done using a procedure called *Splitter Directed Routing*, which will be referred to as SDR in future references. Since this is a crucial component of our algorithm, we describe it in more detail in the appendix.

4. These steps are applied recursively until the size of the subproblems is no more than $\log^2 n$.

Although the original analysis showed that a $\tilde{O}(\log n)$ buffer size may be required, the more recent results on routing enables one to make do with a constant amount of storage in each buffer [21]. The overall running time of the sorting algorithm was analyzed using the property that the problem size at the ith stage of recursion is no more than $n^{\epsilon_0^i}$, $\epsilon_0 < 1$, and there are at most a polynomial number of problems at any stage of recursion. This recursive algorithm can be modelled as a process-tree where the root of this tree corresponds to the original problem and the leaf nodes correspond to procedures that are solved directly. The following result (Reif and Sen [16]) will be used to bound the running time of such algorithms.

LEMMA 2.2. *Given a process-tree that models the recursive algorithm, which has the property that a procedure at depth i from the root takes time T_i such that*

$$P[T_i \ \geq \ k(\epsilon_0)^i c\alpha \log n] \leq \ 2^{-(\epsilon_0)^i c\alpha \log n},$$

all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time.

We shall also use a straightforward generalization of the above result where the process tree is modified in the following manner. Instead of all the subroutines from a node proceeding independently, all the subroutines for a fixed (constant) depth subtree are required to finish

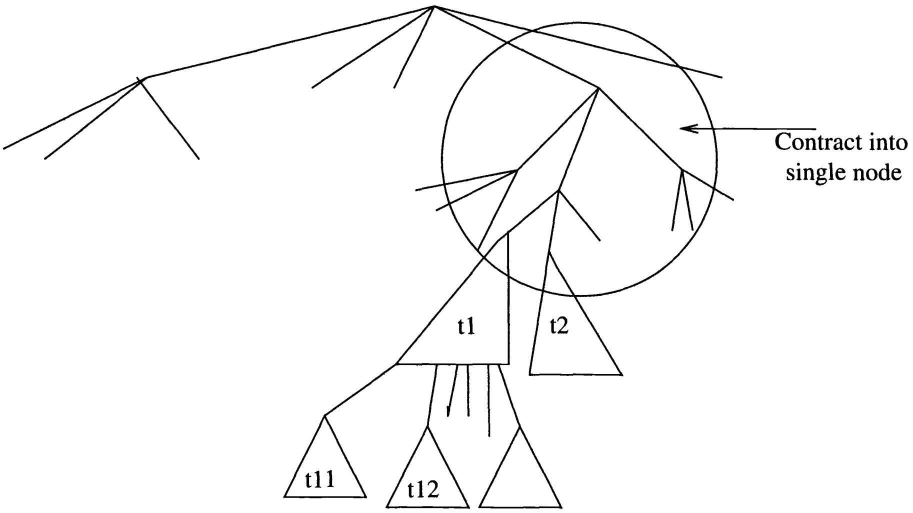


FIG. 2. *By contracting subtrees of a fixed depth, we get another tree satisfying the preconditions of Lemma* 2.2.

before proceeding to the next level of subtrees. This can be reduced to the previous case if we contract a subtree of fixed depth (see Fig. 2) into a single node of the tree. The time taken at any node of this new (contracted) tree also satisfies the property required by Lemma 2.2 with appropriate adjustments in the constants. This follows from the observation that at any level of the process-tree all the procedures terminate in $\tilde{O}(\log n)$ time, and if the process-tree is of constant depth, then clearly the total time is also $\tilde{O}(\log n)$. Thus any contracted node that is actually a subtree of constant depth satisfies the precondition of Lemma 2.2. Consequently, we can prove the following result on similar lines as the previous lemma.

COROLLARY 2.3. *All the leaf-level procedures of this modified tree will terminate in* $\tilde{O}(\log n)$ *steps.*

The above two results will be used repeatedly in the rest of the paper.

3. Binary search without concurrent reads. One of the frequently encountered problems in the case of sequential and parallel algorithms is that of doing a binary search on a tree structure. In particular, given a binary tree of depth $O(\log n)$ whose leaves represent certain intervals and $O(n)$ keys with one processor per key, we would like to locate the interval containing the key in $O(\log n)$ parallel time for each key simultaneously. This problem is trivial in a model allowing concurrent reads. However, the problem becomes more complicated when concurrent reads are not permitted in the model, which is the case with interconnection networks. The simple algorithm uses concurrent reads in an inherent fashion, and for situations where the distribution of the keys is not known, this problem appears even more formidable. To make the exposition simpler we shall first describe a scheme for the *EREW* PRAM and then modify it for the butterfly network. We also note that in certain cases where the intervals and the keys are chosen from the same total ordering the problem reduces to that of merging, which can be done efficiently. However, we are concerned about cases where the intervals may induce an ordering different from the ordering among the keys (see Fig. 3).

We shall look at a special case where the number of leaves is n^{ϵ} for $0 < \epsilon < 1$ and the search tree is roughly balanced, so it has depth $O(\log n)$. The basic strategy is the following. We first try to get a reasonably accurate estimate of the number of keys associated with each of the leaf nodes. Following this, we simulate an interconnection network like the butterfly and allocate an appropriate number of subnetworks based on the estimate. Next we route the keys to their destination subnetworks using a scheme similar to the *SDR* used in *Flashsort.*

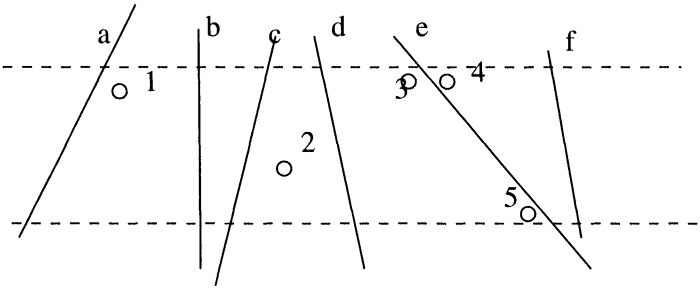


FIG. 3. *Points* 4 *and* 5 *are in different relative orderings with respect to each other and the segments.*

The analysis for *SDR* carries through in this case, although some of the nodes of the splitter tree may be artificial, i.e., both edges departing from a node may lead to the same interval (which is a large subnetwork in our case). We show that each of these steps succeeds with high probability.

To obtain an estimate of the number of keys in each range, we make use of a technique used earlier in Reif and Sen [16] and Rajasekaran and Reif [14]. The result can be summarized in the following manner. Unless otherwise mentioned, we shall use the notation $|S|$ to denote the cardinality of set S.

LEMMA 3.1. *Let S_j, $1 \leq j \leq m \leq n$, be disjoint subsets such that $\sum_{i=1}^{m} |S_i| = n$. If we sample elements independently and uniformly from the union of the subsets with probability p and n_j is the number of elements in the sample from S_j, then for all j such that $p|S_j| \geq d \log n$, n_j is within a constant factor of $p|S_j|$ with high probability. Here d is a constant and the ratio of n_j to $|S_j|$ depends on d.*

In the context of our problem each subset S_i is an interval and $|S_i| = N_i$, the number of keys in the ith interval. As a consequence of this lemma, we set aside $\max\{d \log n \cdot 1/p, E_j\}$ space for each interval. E_j is an estimate for N_j computed from n_j and is $O(1/p \cdot n_j)$. Note that for most applications we choose a value of p such that $md \log n/p \leq n$. We start by selecting each key with probability $1/n^{1-\beta}$; the exact value of β will be determined later. From Chernoff bounds, it follows that the number of keys in the sample is $O(n^{\beta})$ with high probability. For each of these keys, we can determine to which interval they belong by using a brute-force method; for example, simply checking each key against every interval. This can be done in $O(\log n)$ time if we choose $\epsilon + \beta < 1$. From the previous lemma, it follows that if N_i exceeds $cn^{1-\beta} \log n$, where c is a constant independent of n, we can get estimates E_i (of N_i) within a constant factor. So we can set aside $\max(cn^{1-\beta} \log n, E_i)$ space for each interval. For $n^{1-\beta} \cdot n^{\epsilon} \log n < n$, the total space can be bounded by kn for some constant k. From the previous two inequalities involving ϵ it can be seen that $\epsilon < 0.5$. A possible choice for the parameters ϵ and β can be 0.49 and 0.5, respectively. After routing (i.e., by simulating a bounded queue butterfly network), we have the keys in the appropriate interval. We can now sort them in $\tilde{O}(\log n)$ time and determine exactly how many keys there are in each interval.

3.1. Binary search and splitter-directed routing. From the scheme described in the previous paragraph, we set aside $\max(cn^{1-\beta} \log n, E_i)$-sized subnetwork for keys in interval i. The number of rows allocated to a particular interval j is E_j/L. The total number of packets that arrive at any fixed row (that is over all the L ranks) is $\tilde{O}(L)$. This follows from Chernoff bounds as the expected number of packets arriving at any fixed row is $L = \Theta(\log n)$).

Remark. Note that the "subnetworks" are not isomorphic to "butterfly" networks of that size — they have to be routed to "sub-butterflies" after the termination of the splitter directed routing when we map this algorithm to the butterfly network.

We can assume $cn^{1-\beta} \log n$ divides E_i, so we can refer to this size as a "unit" of a subnetwork. We can also assume for simplicity that this is a power of two. Therefore, we can allocate a number of "units" of subnetworks from our estimates. As seen from Fig. 4 the number of these subnetworks may not be aligned with the binary search data structure. Henceforth, any reference to the *search data structure* will imply the splitter tree (of relevant keys) for conducting SDR. More specifically, at a particular node, there could be subnetworks allocated for a particular interval on both the left and right subtrees. We can handle this problem as follows. Each packet i that could go either left or right goes left with probability $L_{k(i)}/(L_{k(i)} + R_{k(i)})$ where $L_{k(i)}$ ($R_{k(i)}$) is the number of "units" on the left (right) subtree for the interval $k(i)$. The packet goes right with the complementary probability. It must be clear that any node of the splitter tree can partition at most one interval and will be labeled accordingly. A packet arriving at the node will check if it belongs to the labeled interval; if it does, then it will branch with the probabilities mentioned previously. The probabilities for this branching can be assigned when the data structure is set up. Setting up the data structure involves constructing the splitter tree as in [18]. The label of each internal node can be determined easily from the number of "units" of each interval. The number of packets going left (right) is a binomial random variable with mean at least $n^{1-\beta} \log n$. Thus the probability that it deviates from the mean by a small factor like $1/\log n$ is less than $1/n^{\alpha}$ for any fixed $\alpha > 0$ from Chernoff bounds. This imbalance factor can accumulate over the $O(\log n)$ branches through which a packet passes during the SDR. However, since $(1 + 1/\log n)^{\log n} = O(1)$, we can conclude that in the modified SDR, the deviation in the number of packets arriving in a subnetwork from the usual SDR is negligible with high probability.

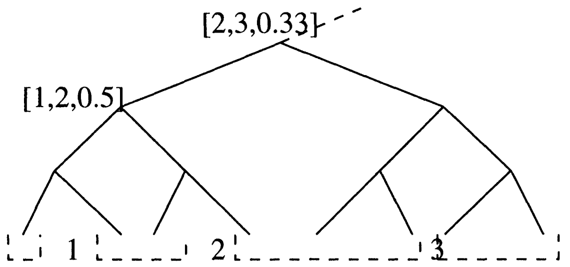


FIG. 4. *Intervals* 1–2 *and* 2–3 *are not aligned with the binary tree. The labels at the nodes indicate the range and the probability that a key in the range taking the left branch. Keys outside this range take the conventional routes.*

We can now use arguments similar to Maggs [12] to bound the congestion. The number of packets that enter any sub-butterfly is within a constant factor of the size of the sub-butterfly. For a switch at rank l, at most $O(L \cdot 2^{L-l})$ rows of the butterfly can be reached for a BF_L. Moreover, a switch at rank l can be reached from 2^l different inputs. If a packet begins at a random node, the probability of reaching a particular switch is 2^{l-L}. The number of packets that pass through a given switch is a binomial random variable. Using Chernoff bounds it follows that the probability of this number (i.e., the congestion) exceeds $O(L)$ and ($\geq O(\log n)$) is less than $n^{-\alpha}$ for any fixed α. We can therefore conclude from the remark after Lemma 2.1 that this *Modified Splitter Directed Routing* (to be referred as $MSDR$) terminates in $\tilde{O}(\log n)$ steps.

We now apply this procedure recursively in the following manner: For each interval with N_i keys we make $\lfloor N_i/n^{1-\epsilon}\rfloor$ "copies" of the subtree whose root will be denoted by L_i. This subtree is present in the (global) shared memory and can be accessed directly. L_i is the internal node of the tree corresponding to interval i. The "copying" can be done easily in $O(\log n)$ phases by the usual doubling strategy — each phase requiring constant time. The number of processors allocated to each of these problems is $n^{1-\epsilon}$. For the remaining keys, say r_i, we round it to the nearest power of 2 and "copy" a subtree with that many nodes and apply this strategy recursively. Clearly, the problem size is decreasing as $n^{1-\epsilon}$, and it follows from Lemma 2.2 that the entire procedure terminates in $\tilde{O}(\log n)$ time. Notice that when the problem size becomes $O(\log n)$, we can solve the problem in $O(\log n)$ time by pipelining a sequential search algorithm. The number of processors required at any stage of the algorithm is no more than twice the number of keys. Thus we may conclude:

THEOREM 3.2. *Given a binary search tree with $O(n)$ leaves and a set of n keys, we can locate the interval (leaf) containing each of the n keys in $\tilde{O}(\log n)$ time using n processors in an EREW* PRAM.

Mapping this algorithm on the interconnection network is not straightforward since we do not have the luxury of allocating the required space as in the case of PRAMS. Instead, we have to simulate a larger sized butterfly network (larger by a constant factor). Moreover, between successive recursive stages we have to do a careful routing to set up the search tree. Unlike the PRAM model where the search tree could be accessed directly by all processors, here we may be forced to conduct global routing to access the relevant parts of the search data structure. So, each recursive call could take $\tilde{O}(\log n)$ time. Note that the sorting algorithm is randomized. Consequently, the expected running time increases by an $O(\log\log n)$ multiplicative factor.

To circumvent the above difficulty we take a different approach. Once the problem size becomes $[O(\log^p n),\ O(\log^p n)]$ (the number of keys and the size of data structure, respectively), we can solve the problem by emulating PRAM algorithms in an additional $O(\log\log^2 n)$[1] deterministic time per step. Here p is some fixed integer. This has the following consequence: we can look at a pruned search tree of size $O(n/\log^p n)$ since if we can determine the subtree where each key belongs to, in an additional $O(\log\log^3 n)$ time, we can complete the entire search procedure. We need some more observations.

Given a search tree T of size $|T|$, we prune off the tree at a certain depth and perform a search on this truncated tree for a set of N keys. We allocate space to the subtrees that are not empty (i.e., there are keys that lie in the corresponding interval) in the following manner. If the keys exceed the subtree size, then we make copies of the subtree and assign a matching number of keys to each copy of the subtree. For the remaining keys we simply make one copy of the subtree. Note that a subtree refers to a part of the original search data structure. Then

LEMMA 3.3. *The total size of the subproblems is no more than $|T| + N$. Size refers to the quantity maximum {size of data structure, number of keys}.*

Proof. For the subtrees that are full, we can charge the space to the keys in it. For the partially full subtrees, there can be at most one for each subtree, and the lemma follows. □

This lemma implies that after completing the search on the $O(n/\log^p n)$-sized tree, $p > 1$, the total space required is less than $2n$, which can be simulated on the network with only a constant factor increase in running time. Moreover, the lemma also says that if we use the above processor allocation strategy, the size of the network required at stage i of the recursion can be bounded as follows:

LEMMA 3.4. *The total size of the subnetworks at stage i is no more than $i(N + |T|)$, where N is the number of keys and T is the data structure.*

[1] We shall use $\log\log^k n$ to denote $(\log\log n)^k$.

Proof. We prove it using induction on the depth (stage) of recursion i. Clearly this holds after the first stage ($i = 1$) from Lemma 3.3. Assume that this is true for stage k, i.e., the total space required is no more than $k(N + |T|)$. In the next recursive call, we use the strategy described previously for making "copies" and use the following charging scheme. For the "completely full" subtrees of stage k, the space required for the completely full subtrees of stage $k + 1$ is charged to the keys. For the partially full subtrees of stage $k + 1$ resulting from "completely full" subtrees of stage k, the extra space can again be charged to the keys since the size of the data structure in stage k was matched with the number of keys. For the partially full subtrees from stage k, the completely full subtrees can be charged to the keys by the previous argument. For the partially full subtrees of stage $k + 1$ resulting from partially full subtrees of stage k, they are charged to the data structure of stage k.

Hence the total amount of extra space required at stage $k + 1$ is $N + |T|$. □

From the above observations, we proceed as follows. We consider first a reduced problem where the number of keys $N = n/\log n$ and the size of the tree is $n/\log^3 n$. The $O(n/\log n)$ keys are chosen uniformly at random, and the tree is pruned to this size. The processor-allocation strategy at recursive stage j is to allocate subnetworks proportional to size $(\log\log n - j)(|T_i| + n_i)$ for a subproblem with subtree size $|T_i|$ and n_i keys. We postpone the discussion of processor allocation to the next section; however, we make note of the following fact.

Since the $|T_i|$'s are identical and $n_i \leq |T_i|$, the network can be partitioned into equal-sized subnetworks proportional to $|T_i|$. The motivation for this is that such a partitioning can be done relatively easily (cf. Lemma 3.5). The drawback is that the space requirement could grow by a factor of two at each recursive level (Lemma 3.3), so we set aside space proportional to $2^{\log\log n - j}|T_i|$ at stage j. Since $\sum_i |T_i|$ can be bounded by the total number of keys, i.e., $n/\log n$, the space requirement is $O(n)$. Even if the number of recursive levels is greater than $\log\log n$ (but still $O(\log\log n)$), we can conduct a global routing after every $\log\log n$ stages so that the size is no more than $2n$ after each of those processor-allocation procedures.

From the previous lemmas, we have sufficient processors to carry out the reduced search problem. If the keys are chosen uniformly at random with probability $1/\log n$, then for all the subtrees of size $\log^3 n$ that have more than $\log^2 n$ keys we have very accurate estimates (Lemma 3.1). So we can set aside $\max\{c\log^2 n, E_l\}$-sized subnetworks for subtree l where E_l is an estimate. Note that $cn/\log n + \sum_l E_l$ is $O(n)$ so we can perform the SDR on all the n keys this time. Subsequently we count explicitly the number of keys in each subtree and allocate subnetworks with the maximum problem size of $O(\log^3 n)$. We can then emulate any PRAM algorithm adding $O(\log\log^3 n)$ to time complexity.

3.2. Load balancing and processor reallocation. The previous strategy for processor allocation would work if the size of the subproblems always matches the sub-butterfly size; however, it may not always be the case. So we have to design a more general processor-allocation procedure that can be done dynamically and also evenly distribute the work load among the sub-butterflies. If all the subproblems are of the same size, then such a load distribution can be achieved easily.

LEMMA 3.5. *Given problems p_i of identical sizes, such that $\sum |p_i| \leq N$, where N is the size of the butterfly network, the problems can be allocated to sub-butterfly networks $\mathcal{N}_i$ satisfying $c|\mathcal{N}_i| \geq |p_i|$. Here c is a constant less than* 6.

Proof. Let p be the largest number less than or equal to p_i that is of the form $k \cdot 2^k$ where k is an integer. For $p_i > 2 \cdot 2^2$, the ratio p_i/p is less than 3. Let $N = h \cdot 2^h$ and $p = l \cdot 2^l$. Then from Fact 1, the butterfly network can be rank-partitioned into $\lfloor h/l \rfloor$ parts, each consisting of 2^{h-l} sub-butterflies of size p. Assuming that $h/l > 1$, the "unused" part of the butterfly is less than 1/2, where by "unused" part we are referring to the $h - (\lfloor h/l \rfloor) \cdot l$ ranks. To each sub-butterfly of size p in each partition we allocate 2 problems of size $|p_i|$. This achieves the required ratio as stated in the lemma. □

The previous lemma in conjunction with Fact 2 guarantees that if all problems are of the same size, then the processor allocation achieves roughly the same work-load distribution as in PRAM (as is the case with binary search). However, it is not clear as to how to do the same when the problems may be of varied sizes. Instead, we aim for a weaker result than Lemma 3.5.

Let us denote the subproblem j by $\mathcal{P}_j$ and a subnetwork (which may not be topologically equivalent to a butterfly network) to which the subproblem is allocated by $\mathcal{N}_j$.

LEMMA 3.6. *Given that $\sum |\mathcal{P}_j| \leq c_1 N$, where N is the total size of the network and c_1 constant, there is a processor-allocation scheme such that for all j, $|\mathcal{P}_j| \leq c_2 |\mathcal{N}_j|$ where c_2 is another constant. Moreover, if some algorithm $\mathcal{A}$ takes time $O(\log^\alpha |\mathcal{P}_j|)$, $\alpha > 1$, for problem $\mathcal{P}_j$ on a butterfly network, then the allocation scheme ensures that the algorithm running in parallel for all the subproblems terminate in $O(\log^\alpha N)$ time.*

Note that the running time for algorithm $\mathcal{A}$ depends on N and not simply on $|\mathcal{P}_j|$ as in the case of PRAM.

Proof. Assume that the network is of size $M \geq k \sum |\mathcal{P}_j|$ since we can always simulate a network larger by a factor of k. This constant k will compensate for the constant c_1 and simplify the proof while slowing down the network only by a constant factor. Let $N = \hat{h} 2^{\hat{h}}$ and $M = h \cdot 2^h$. Let $S_i, i \geq 0$, denote the set of subproblems whose sizes are in the range $\lfloor h/2^i \rfloor 2^{\lfloor h/2^i \rfloor}$ and $\lfloor h/2^{i+1} \rfloor 2^{\lfloor h/2^{i+1} \rfloor}$. For notational simplicity we shall denote $\lfloor h/2^i \rfloor$ by h_i. Note that $h_i / h_{i+1} < 4$ for $h > 4$. Form groups by sorting the sizes of the problems.

The idea is to allocate contiguous rows of the network to each S_i. For problems in S_i, we make $m_i = \lfloor h/h_i \rfloor$ horizontal partition in the network, each of h_i ranks. In S_i let $k_{i,j}$ denote the set of problems of sizes between $(h_i - j) \cdot 2^{h_i - j}$ and $(h_i - j - 1) \cdot 2^{h_i - j - 1}$. Let $r_{i,j} = |k_{i,j}| \cdot 2^{h_i - j} / m_i$ rounded to the nearest integer, which is the number of rows allocated to problems of $k_{i,j}$. Allocate two problems to each $h_i \times 2^{h_i - j}$ "rectangular partition."

At most 1/2 the number of ranks is unused when m_i horizontal partitions are created. At most 3/4 of the space within each rectangular region is unused since $h_i / h_{i+1} < 4$. Moreover, for each $i = 1, 2, \ldots, \hat{h}$, at most 2^i rows are unused viz., $h \cdot 2^{\hat{h}+1}$ processors. In addition, we may have to double the number of rows so that $\sum_i k_{i,j}$ add up to a power of two for the butterfly network. From the previous bounds on the amount of "unused" space, one can verify that by choosing $k > 64$, the above scheme succeeds.

Note that for each problem size within S_i only the bottom-most h_i ranks form a sub-butterfly. So for problems in S_i, we require m_i times the time required to execute an algorithm $\mathcal{A}$. For logarithmic time complexity of $\mathcal{A}$ the total time in each S_i is clearly $O(\log N)$. A similar observation applies to the case $\alpha > 1$. □

The above two lemmas do not directly imply a general processor-allocation strategy. Even in the simple case where all problems are of the same size, if we were to continue with this scheme recursively, a network could be loaded by a factor of approximately c^2 in the next recursive call. By "loading" we mean the ratio of the subproblem to the subnetwork size (which is unity in the beginning), and this is also a lower bound on the slow down of the algorithm. This is undesirable if the depth of recursion is $O(\log \log n)$. However, we can use the previous lemmas in the following useful manner. We can apply them for a constant number of stages of recursive calls without increasing the run time by more than a constant factor (in every call the load on a processor increases by a constant factor). We shall show that this suffices for our processor-allocation strategy, where we redistribute the load "evenly" after a fixed number of stages of recursive call. The network topology, where the subnetworks are of sizes $h2^h$ (h is an integer) necessitates this kind of rebalancing strategy.

We shall start with a special case where the network size is $h2^h$ where $h = 2^l$ for some integer l. In two stages of recursive call we can reduce the subproblem sizes to $2^{h/2}$. One call is not sufficient since the subproblem sizes could be larger than $n^{1/2}$, which is $2^{h/2+\log h}$ in this case. Now we can pack $h/2$ subproblems into a subnetwork of size $(h/2) \cdot 2^{h/2}$. Moreover, there are $2 \cdot 2^{h/2}$ subnetworks of this size (Fact 1), which implies that the entire network is being used instead of a fraction of it. Now we are in subnetworks of size $(h/2) \cdot 2^{h/2}$, and hence the procedure can be applied inductively. Notice however that we have $h/2$ subproblems in the network (instead of one) and we have to use Lemma 3.6 to allocate subnetworks, thereby loading the subnetwork by a constant factor in the next recursive call. But this is only for a fixed (at most 2) stages of recursive call, and hence we can maintain the same asymptotic run time. Moreover, we also observe the following.

LEMMA 3.7. *If a set of processors is initially loaded by a factor λ, then this load factor does not increase during the course of the processor-allocation strategy.*

To generalize this scheme to an arbitrary value of network-size, we reduce the problem sizes to 2^{2^k} for some k. This can be done in a constant number of recursive calls since $2^{2^{k+1}} = (2^{2^k})^2$. Using Lemma 3.6, we can allocate subproblems such that only a constant fraction of the network is unused. This happens if 2^k does not divide h evenly, and consequently some processors may be loaded by a factor of 2. But from this stage we can use the scheme described in the previous paragraph without increasing the load factor any further (Lemma 3.7).

We summarize our result as follows.

THEOREM 3.8. *Given a binary search tree with $O(k)$ leaves, we can perform binary search for $O(k)$ keys in $\tilde{O}(\log k)$ time in a butterfly network using k processors.*

In a more general scenario, as will be required for the applications in the next section, the subproblems could be of different sizes. However, a key fact is that we can bound the size of the maximum-sized subproblem by $n^{1-\epsilon}$ where n^{ϵ} is a sample size (similar to the previous case). Let $[i]$ denote the largest number less than i that is a power of 2, i.e., if $[i] = 2^j$, then $2^j \leq i \leq 2^{j+1}$. By choosing the sample sizes appropriately (n^{ϵ} for some $\epsilon > 0$), we can reduce the maximum subproblem size to $2^{[h]}$. This may require more than one recursive call, each of which is handled using the processor-allocation scheme of Lemma 3.6.

We then greedily pack as many subproblems as we can in a subnetwork of size $[h] \cdot 2^{[h]}$. This is done by sorting the subproblem sizes and an application of parallel prefix followed by actually routing the subproblems to the assigned subnetworks. We pack in subproblems as long as there is space, so that we do not "load" a subnetwork by more than a factor of $(1 + 1/\log n^{1-\epsilon})$ (specifically, $1 + 1/h$ since we are packing at most $h + 1$ subproblems in a network that can accommodate h subproblems).

We apply this procedure recursively such that the extra loading factor at stage i of recursion is no more than $1 + 1/\log n^{(1-\epsilon)^i}$. Note that in the first stage of recursion, we could misuse a constant fraction of the network (because of problem size mismatch), but thereafter the degree of misuse can be bound by the above quantity by our processor-reallocation strategy. If the subproblems are never smaller than $\log n$, then the load factor can be bound by $\prod_{i=1}^{O(\log\log n)}(1 + 1/\log n \cdot (1 - \epsilon)^i)$. If the smallest term is larger than $\log\log n$, then the above product is bounded by e^{α}, where α is a constant. Note that the "load factor" is maintained inductively in the subnetworks, by redistribution of work load. The slowdown at any stage is at least 2 and no more than e^{α}.

4. Applications to computational geometry. In this section we apply the techniques developed in the previous sections to map some geometric algorithms efficiently on the butterfly network. The description relies partly on the work of the authors on randomized PRAM algorithms. For a better understanding of the original algorithms, the reader is encouraged to refer to the papers [17], [16].

4.1. Searching in arrangements. We focus our attention on the following problem. Given an arrangement of n^{γ} lines, $\gamma < 1/3$, we want to find out for n given points the region to which it belongs. In particular, we would like to do this in $O(\log n)$ time on a butterfly network. Dobkin and Lipton had described a very simple method for solving this problem using the following data structure. Find out all the pairwise intersections (there are $n^{2\gamma}$ in this case) and project them on the X-axis. Within each interval, the lines can be totally ordered and one can set up a binary tree corresponding to this ordering. Thus, there are $n^{2\gamma}$ binary trees each of size n^{γ}. To find the region (which is a trapezoid by the above partitioning scheme) to which a point belongs, we do two binary searches — one along each direction.

To implement this algorithm in parallel, the data structure can be set up very easily in parallel by sorting. To search n points, we first sort the n points by their x coordinates and merge them with the endpoints of the intervals. This saves the binary search in the x-direction. If I_i is the set of points in the ith interval, we allocate a subcube of size $|I_i| + n^{\gamma}$ nodes. This is done using the network partitioning scheme of Lemma 3.6. Since $n + n^{3\gamma} < 2n$, for appropriate choice of γ and moderately large n, this can be done by simulating a network twice the size. The binary search can then be done within each subnetwork in $\tilde{O}(\log n)$ time using the procedure outlined in the previous section. Thus the overall algorithm runs in $\tilde{O}(\log n)$ time.

THEOREM 4.1. *Given n points on the plane and an arrangement of m lines, $m < n^{1/3}$, for each of the n points, the (unique) region of the arrangement containing the point can be identified in $\tilde{O}(\log n)$ in an n-node butterfly network.*

We shall see in the next section that these trapezoidal regions define a finer partition of equivalence classes, which are the regions in the original partition.

4.2. A framework for randomized divide-and-conquer. Given a set of nonintersecting line segments and points, we wish to determine for each point, the line segment(s) that lay directly below and above each of these points. There may be 0, 1, or 2 such edges, which are called the trapezoidal edges.

We shall recapitulate the main steps of the algorithm for trapezoidal decomposition described in [20] in a more general context of a divide-and-conquer algorithm.

1. Select $O(\log n)$ subsets of random objects (in case of two-dimensional hulls these were half-planes) each of size $\lfloor n^{\epsilon} \rfloor$ for some $0 < \epsilon < 1$. Each such subset is used to partition the original problem into smaller subproblems. A sample is "good" if the maximum subproblem size is less than $O(n^{1-\epsilon} \log n)$ and the sum of the subproblem sizes is less than $\bar{c}n$ for some constant $\bar{c}$. From the probabilistic bounds proved in Reif and Sen [17] (for problems considered in the paper) and Clarkson [7] (for very general situations that are applicable to our problems), it is known that the first condition for a "good" sample holds with high probability. However, the second condition is satisfied with probability at least 1/2.

2. Select a sample that is "good" with high probability using *Polling*. At least one of the $\log n$ samples in the previous case is "good" with high probability. *Polling* [16] is a sampling technique that allows us to choose a "good" sample efficiently.

3. Divide the original problem into smaller subproblems using the "good" sample. The maximum size can be bound by $O(n^{1-\epsilon} \log n)$.

4. Use a *Filtering* procedure to bound the sum of the subproblem sizes by some fixed measure like the output size or input size. The reason for this being that the probabilistic bounds in step 1 bounds the sum of the subproblems by $\bar{c}n$. If this increase by a multiplicative constant continues over each recursive stage, after $O(\log \log n)$ depth, the input size will have

increased by a polylogarithmic factor. This *filtering* procedure is problem dependent and uses the specific geometry properties of a problem. The purpose is to bound the number of processors.

5. If the size of a subproblem is more than a threshold (usually it is chosen to be $O(\log^k n)$ for some constant k), then call the algorithm recursively, else solve the problem using some direct method.

The algorithms presented in this paper are based on this approach. However, the implementation of some of these steps depend heavily on the specific problem. The procedure used for dividing the subproblems depends on the problem at hand, and this is also closely linked with *Polling*. *Polling* involves selecting $n/\log^2 n$ input objects and partitioning them using a random subset chosen in step 1 instead of the entire input set. Since there are $O(\log n)$ subsets, this saves the extra work we would have had to do if we tested the "goodness" of the sample on the entire input. The *Polling lemma* [16] guarantees that with high probability we can choose a good sample using this method. Since the test for "goodness" is carried out independently for each of the samples, this part of the algorithm is inherently parallelizable even on the networks. Perhaps the step that is most specific to a problem is the *Filtering* step where we have to use some geometric properties of the problem.

In the context of mapping algorithms to interconnection networks, step 3 turns out to be the most difficult. Steps 1 and 2 are inherently parallel, and step 4 for the problems that we are interested in, is not very involved. We shall show that step 3 reduces to searching in an arrangements of lines in two dimensions.

4.3. Trapezoidal decomposition. For trapezoidal decomposition, we sample line segments and build its convex map (see Fig. 5). We can build this using the following brute-force approach. For every segment endpoint, we order the line segments by their y coordinates. For every segment, we order the projection of the endpoints those are visible from below and above. From this information, we can construct the trapezoids by simulating "pointer-jumping" a fixed (at most 6) number of times.

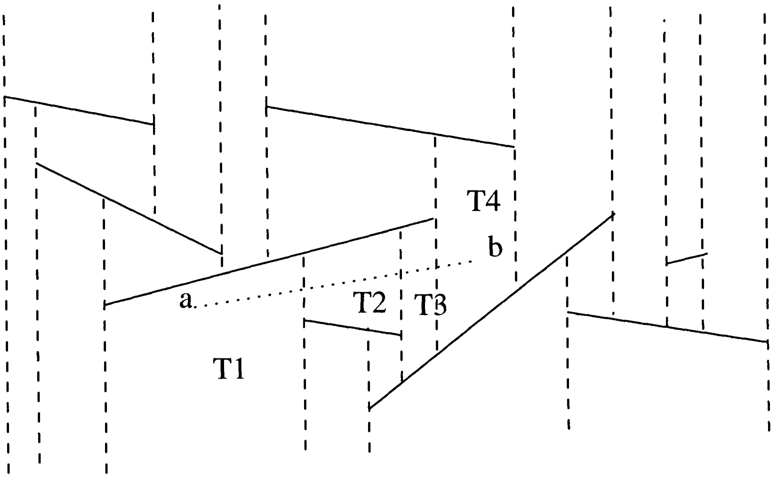


FIG. 5. *Convex map of nonintersecting line segments. Line segment ab intersects the trapezoids T1, T2, T3, T4.*

LEMMA 4.2. *The trapezoidal map of n^ϵ segments can be constructed in $\tilde{O}(\log n)$ time in an $n^{3\epsilon}$ processor butterfly network.*

The remaining segments (not part of the sample) are partitioned into subproblems defined by the trapezoidal map. The partitioning step in trapezoidal decomposition can also be reduced to the problem of searching in arrangements of linear constraints in two dimensions (see Reif and Sen [17]). This is also known as the *locus* method where the problem reduces to finding the region of the arrangement in which a query point lies. These regions are preprocessed so that after the point location one has to perform a table look-up to determine the answer. In this case it is a set of trapezoids that a segment intersects. Since this can be of various lengths, it has to be done with a little care. First the number of trapezoids is determined, which is an integer, and then an appropriate number of processors is delegated the responsibility of determining the actual trapezoids. This allocation is done with the help of a prefix computation. Subsequently, each of these processors do a table look up. The table look-up is done by emulating a single step of the CREW PRAM in $\tilde{O}(\log n)$ steps.

For the filtering step, we need to keep track of parts of segments that completely span a trapezoid. Such segments within a trapezoid have to be processed for binary search such that for endpoints lying within the trapezoid we can quickly determine its closest visible (upper and lower) segments. This can be done in $\tilde{O}(\log n)$ time using the binary search algorithm described in § 3. Moreover, only the segments that partially or completely lie within a trapezoid are needed for further recursive calls.

At each stage we keep track for each endpoint, which are its closest (upper and lower) segments, and hence at the end we have its trapezoidal edge(s). From this information, we can decompose a simple polygon into one-sided monotone polygons (see Fig. 7). This can be accomplished by an application of sorting and prefix computation (Goodrich [9]). Using Yap's [23] technique, further calls to trapezoidal decomposition within these one-sided monotone polygons enable us to determine all the triangulation edges. The procedure is as follows.

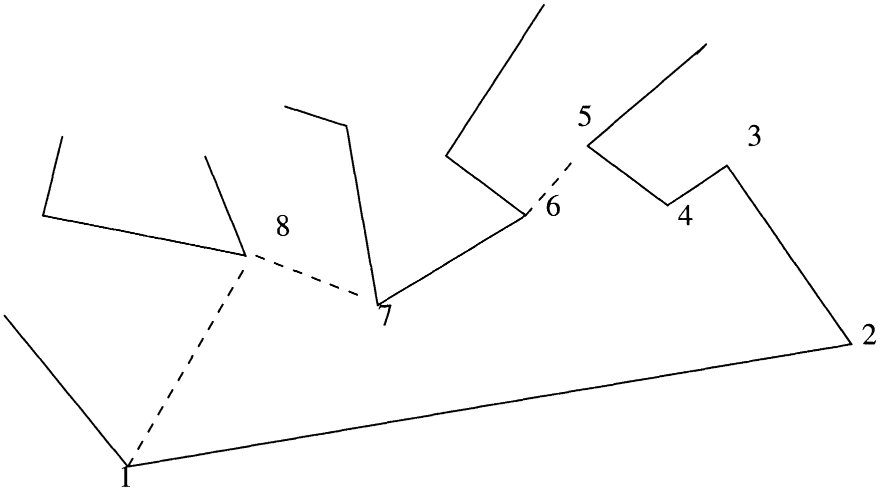


FIG. 6. *Decomposition into one-sided monotone polygons. The dotted lines indicate some of the triangulation edges. The distinguished edge is* 1, 2.

Step 1. Construct the horizontal trapezoidal decomposition of the one-sided monotone polygon, i.e., for every vertex v determine the edges of the polygon through which a horizontal line passes v while the line is contained within the polygon. Assume that the distinguished edge, i.e., the one with which the polygon is monotone, is horizontal.

Step 2. Denote the left visible edge by $l(v)$ and right visible edge by $r(v)$, and let $\lambda(v)$ and $\rho(v)$ denote vertices of $l(v)$ and $r(v)$, respectively, which have the lower altitude.

Step 3. The set of edges $E = \{v\lambda(v)\} \cup \{v\rho(v)\}$ form a triangulation of the polygon.

Trapezoidal decomposition also enables us to solve the problem of determining visibility of a set of nonintersecting line segments when they are projected orthogonally. Sort the endpoints of the line segments projected on the x-direction and choose a point (say the midpoint) in every interval. For each of these points, determine its trapezoidal edge using the trapezoidal decomposition algorithm described above.

We can state the main result of this section as the following.

THEOREM 4.3. *We can construct the trapezoidal decomposition of n segments in time $\tilde{O}(\log n)$ on an n-processor, bounded-buffer butterfly network, where n is the input size of the problem. Using this as a subroutine, a simple polygon of n vertices can be triangulated in the same bounds.*

To the best of our knowledge, for the above problems, no matching algorithms are known even on the *EREW* PRAM model.

4.4. Two-dimensional convex hulls. We need the following preliminary results:

LEMMA 4.4. *The convex hull of n^α, $\alpha < 1/2$, half-planes can be constructed in $O(\log n)$ time in an n-node butterfly network.*

Assume that the intersection is nonempty and bounded, and then a brute-force algorithm suffices.

Given this hull and a point that lies in the final hull, we divide this hull into triangular sectors and find out for the remaining half-planes which of the sectors they intersect. For this we go to the dual plane and search in the arrangements of the (duals of) vertices of the convex hull of the sampled half-spaces. This can be done using the procedure described in the previous section — the details are straightforward and are omitted for brevity.

The filtering scheme for two-dimensional convex hulls involves an application of two-dimensional maxima. A large number of redundant half-planes like a can be identified by an application of the two-dimensional maxima algorithm where the coordinates are the ranks of the ordering of intersections of the half-planes on the two sides of a sector (starting from the apex). For example, in Fig. 7 the coordinate corresponding to b dominates that of a.

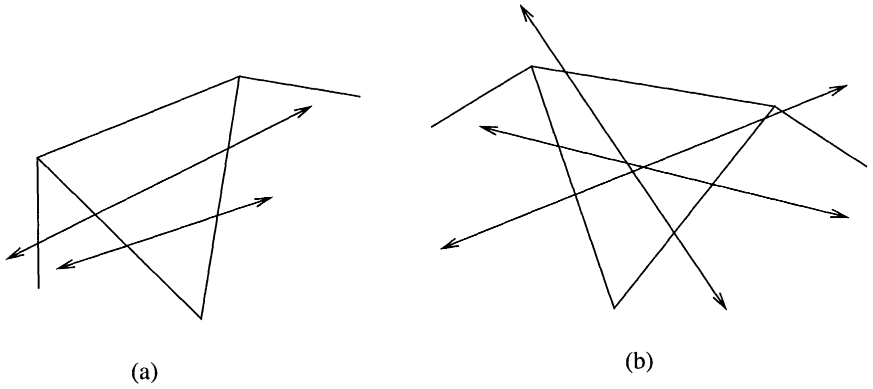


FIG. 7. *In case (a), line a is completely occluded by b. In case (b), line c is occluded by d and e but not by any one of them competely. However, in the other sectors, line c is eliminated by case (a).*

LEMMA 4.5. *The maximal points of n points in a plane can be computed in $\tilde{O}(\log n)$ time in an n-node butterfly network.*

Proof. We use a variation of the sequential algorithm in the following manner. We sort the points based on their x-coordinates and perform a prefix computation on the operation maximum on this sorted set. If $p_1, p_2, \ldots, p_n$ are the points sorted on the x-coordinates, then we compute an array MAX such that MAX_i has the maximum y coordinate for all points p_j such that $j > i$. If the y-coordinate of p_i is smaller than MAX_i, then it is not a maximal element. □

Finally we can state the result of this section as

THEOREM 4.6. *The convex hull of n points in a plane can be constructed in $\tilde{O}(\log n)$ in an n-node butterfly network with bounded buffer size.*

5. Concluding remarks. In this paper, we have described a strategy for implementing PRAM algorithms for geometric problems on fixed-connection networks. These methods involve tackling some of the very basic problems, like binary search and dynamic load balancing that we take for granted in PRAM models. Our techniques use a number of ideas from *Flashsort*, but they have to be modified to handle more difficult situations, namely, searching in partial orders and dynamically allocating subnetworks to recursive calls.

An important goal of our research is to build a hierarchy of fundamental geometric algorithms for fixed-connection networks similar to that of PRAM algorithms. Two important problems in this list include those of constructing two-dimensional Voronoi diagrams and three-dimensional convex hulls in optimal (or near-optimal) time.

6. Appendix.

6.1. Splitter-directed routing. Let X be the set of cN keys that are totally ordered by the relation $<$. V is the set of nodes in the network. Suppose that for some l $(1 \leq l \leq n)$ we are given a set of *splitters* $\Sigma \subseteq X$ of size $|\Sigma| = 2^l - 1$. We index each splitter $\sigma[w] \in \Sigma$ by a distinct binary string $w \in \{0, 1\}^L$ of length less than L. Let $\prec$ denote the ordering defined as follows: For $u, v, w \in \{0, 1\}^L$, $w0u \prec w \prec w1v$. We require that for all $w_1, w_2 \in \{0, 1\}^L, \sigma[w_1] < \sigma[w_2]$ if and only if $w_1 \prec w_2$. We assume that a copy of each splitter $\sigma[w]$ is available in each node $V[w]$. $V[w]$ is the set of nodes with rank $|w|$ with row addresses prefixed by w (same as in Reif and Valiant [18]).

Let $X[\lambda] = X$ where λ is the empty string. Initially we assume that the keys of $X[\lambda]$ are located in $V[\lambda]$, that is, the nodes of V having stage 0. The splitter-directed routing tree is executed in l temporarily overlapping stages $i = 0, 1, \ldots, l-1$. For each $w \in \{0, 1\}^i$ the set of keys $X[w]$ that are eventually routed through $V[w]$ is defined recursively. The splitter $\sigma[w]$ partitions $X[w] - \sigma[w]$ into disjoint subsets

$$X[w0] = \{x \in X[w] | x < \sigma[w]\}$$

and

$$X[w0] = \{x \in X[w] | x > \sigma[w]\},$$

which are subsequently routed through $V[w0]$ and $V[w1]$, respectively.

In our case, we assume that after each recursive call, the subnetworks (of varying sizes corresponding to different subroutine calls) are relabeled as if they were isolated networks. The $V[w]$'s are then defined accordingly. The time analysis for this procedure is carried out using a *delay-sequence* argument, and it was shown [15] that the running time is $\tilde{O}(\log n)$ time in a BF_n.

6.2. Probabilistic inequalities. We say a random variable X upper bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, Prob$(X \leq x)$ $\leq$ Prob$(Y \leq x)$.

A Bernoulli trial is an experiment with two possible outcomes, namely, success and failure. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can easily be seen to be

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}.$$

The tail end of the Binomial distribution can be bounded by *Chernoff* bounds. In particular, the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob}(X \geq m) \leq \left(\tfrac{np}{m}\right)^m e^{m-np}, \tag{1}$$

$$\text{Prob}(X \leq m) \leq \left(\tfrac{np}{m}\right)^m e^{-np+m}, \tag{2}$$

$$\text{Prob}(X \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2), \tag{3}$$

$$\text{Prob}(X \geq (1+\epsilon)np) \leq exp(-\epsilon^2 np/3) \tag{4}$$

for all $0 < \epsilon < 1$.

REFERENCES

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, *Parallel computational geometry*, Algorithmica, 3, (1988), pp. 293–327.

[2] M. Atallah, R. Cole, and M. Goodrich, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM J. Comput. 18 (1989), pp. 499–532.

[3] S. Chandran, *Merging in Parallel Computational Geometry*, Ph.D. thesis, University of Maryland, College Park, MD, 1989.

[4] A. Chow, *Parallel Algorithms for Geometric Problems*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1980.

[5] K. Clarkson, *A probabilistic algorithm for the post-office problem*, Proc. of the 17th Annual STOC, 1985, pp. 174–184.

[6] ———, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[7] ———, *Applications of random sampling in computational geometry* II, Proc. of the 4th Annual ACM Symposium on Computational Geometry, 1988, pp. 1–11.

[8] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[9] M. Goodrich, *Efficient Parallel Techniques for Computational Geometry*, Ph.D. thesis, Purdue University, 1987.

[10] D. Haussler and E. Welzl, *ϵ-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–152.

[11] A. Karlin and E. Upfal, *Parallel hashing — an efficient implementation of shared memory*, Proc. of the 18th Annual STOC, 1986, pp. 160–168.

[12] B. Maggs, *Locality in Parallel Computation*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1989.

[13] R. Miller and Q. Stout, *Efficient parallel convex hull algorithms*, IEEE Trans. Comput., 37 (1988), pp. 1605–1618.

[14] S. Rajasekaran and J. Reif, *Optimal and sublogarithmic time radomized parallel sorting algorithms*, Tech. report, Aiken Computing Lab, Harvard University, Cambridge, MA, 1986; SIAM J. Comput., to appear.

[15] A. Ranade, *How to emulate shared memory*, Proc. of the 28th IEEE FOCS, 1987, pp. 185–194.

[16] J. Reif and S. Sen, *Polling: A new random sampling technique for computational geometry*, Proc. of 21st Annual STOC, 1989, pp. 394–404.

[17] J. Reif and S. Sen, *Optimal randomized parallel algorithms for computational geometry*, Algorithmica, 7 (1992), pp. 91–117.

[18] J. Reif and L. Valiant, *A logarithmic time sort for linear size networks*, J. Assoc. Comput. Mach., 34 (1987), pp. 60–76.

[19] R. Reischuk, *A fast probabilistic parallel sorting algorithm*, Proc. of the 22nd IEEE FOCS, 1981, pp. 212–219.

[20] S. Sen, *Random Sampling Techniques for Efficient Parallel Algorithms in Computational Geometry*, Ph.D. thesis, Duke University, Durham, NC, 1989.

[21] B. Maggs, T. Leighton and S. Rao, *Universal packet routing algorithms*, Proc. of the 29th IEEE FOCS, 1988, pp. 256–269.

[22] L. Valiant, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350–361.

[23] C. Yap, *Parallel triangulation of a polygon in two calls to the trapezoidal map*, Algorithmica, 3 (1988), pp. 279–288.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 652–661, June 1994

012

COMMUNICATION-SPACE TRADEOFFS FOR UNRESTRICTED PROTOCOLS*

PAUL BEAME†, MARTIN TOMPA†, AND PEIYUAN YAN‡

Abstract. This paper introduces communicating branching programs and develops a general technique for demonstrating communication-space tradeoffs for pairs of communicating branching programs. This technique is then used to prove communication-space tradeoffs for any pair of communicating branching programs that hashes according to a universal family of hash functions. Other tradeoffs follow from this result. As an example, any pair of communicating Boolean branching programs that computes matrix-vector products over GF(2) requires communication-space product $\Omega(n^2)$, provided the space used is $o(n/\log n)$. These are the first examples of communication-space tradeoffs on a completely general model of communicating processes.

Key words. communication complexity, lower bound, tradeoff, branching program, universal family of hash functions

AMS subject classifications. 68Q05, 68Q10, 68Q22, 68Q25

1. Communication and space.

The amount of communication required among processors cooperatively performing a computation is often the dominant factor in determining the efficiency of parallel or distributed systems, in both practical and theoretical terms. In addition, communication complexity has found surprising applications in the complexity of Boolean circuits (Karchmer and Wigderson [14], Raz and Wigderson [19]), Boolean decision trees (Hajnal, Maass, and Turán [13]), combinatorial optimization (Yannakakis [23]), VLSI (Aho, Ullman, and Yannakakis [3], Lipton and Sedgewick [16], Mehlhorn and Schmidt [18], Yao [25]), and pseudorandom number generators (Babai, Nisan, and Szegedy [5]).

Nearly all the previous work on the communication complexity of various problems has focused on their communication requirements alone, in the absence of any limitations on the individual processors. Lam, Tiwari, and Tompa [15] initiated the study of communication complexity when the processors have limited work space. As is customary, the systems studied consist of two communicating processors that are given private inputs x and y, respectively, and are to output some function $f(x, y)$. With no restriction on the workspace it is impossible to prove superlinear lower bounds on the amount of communication, since one processor can send its entire input to the other, which then computes and outputs $f(x, y)$. In contrast, Lam, Tiwari, and Tompa proved several nonlinear lower bounds on communication in the straight-line model, when space is limited. For example, one of their results of particular relevance to what follows is that multiplication of an $n \times n$ matrix by an n-vector in the Boolean straight-line model with one-way communication requires communication $C = \Theta(n^2/S)$ when the processors' workspace is restricted to S.

In this paper we remove the restrictions of straight-line computation and one-way communication, proving for the first time communication-space tradeoffs on a completely general model of communicating processes. This result is analogous to Borodin and Cook's time-space tradeoff for sorting on a general sequential model [7].

More specifically, we introduce the notion of communicating branching programs. We use these to demonstrate that if one of the branching programs is given a member h of a universal family of hash functions (Carter and Wegman [9], [10]) and the other is given x

*Received by the editors October 1, 1992; accepted for publication (in revised form) February 17, 1993. This material is based upon work supported in part by National Science Foundation grants CCR-8858799 and CCR-8907960 and by IBM Research Contract 16980043.

†Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195.

‡Mathematics Department, Lycoming College, Williamsport, Pennsylvania 17701.

and their goal is to compute $h(x)$ cooperatively, then their communication C and space S must satisfy the tradeoff $CS = \Omega(nm)$, where h maps n-bit inputs to m-bit outputs, provided $S = o(n/\log m)$. As an example, any pair of communicating Boolean branching programs that multiplies an $n \times n$ matrix by an n-vector over GF(2) satisfies $CS = \Omega(n^2)$, provided $S = o(n/\log n)$. Similar applications hold over more general finite fields and for other hash functions such as arithmetic over large finite fields, convolution, and matrix multiplication.

If a single processor can compute $f(x, y)$ in time C and space S, then a system of two processors can compute $f(x, y)$ in communication $O(C)$ and space $O(S)$, simply by communicating the result of every instruction executed by either. Thus, the lower bounds outlined above imply the corresponding time-space tradeoffs of Grigoriev [12] for straight-line programs and Abrahamson [2] for branching programs. The converse, however, is false. Whereas the time T and space S must satisfy $TS = \Omega(n^2)$ when computing the discrete Fourier transform [2], [21], [27] or sorting [6], [8], [21], Lam, Tiwari, and Tompa [15] demonstrated that both of these functions can be computed in linear communication steps and $O(\log n)$ space simultaneously. Thus, these results strictly generalize previous time-space tradeoffs.

2. Communicating branching programs. The general framework for dealing with problems of two party communication requires an accurate notion of both the computational power of the two parties involved and their method of communicating with each other. A restricted model in which each party executes a straight-line program was defined by Lam, Tiwari, and Tompa [15]. In their model each straight-line program is augmented with *send* and *receive* instructions. They leave open the question of defining an appropriate nonoblivious model.

Since branching programs have proved to be useful sequential models for the simultaneous measure of time and space [1], [2], [6], [7], [8], [27] it is natural to use them to model the communicating parties. Making the analogous changes to branching programs that Lam, Tiwari, and Tompa made to straight-line programs leads to the following model.

A *communicating pair of* (*Boolean*) *branching programs* consists of two branching programs, known as the X-program and the Y-program, that have input vectors $x \in X = \{0, 1\}^{n_X}$ and $y \in Y = \{0, 1\}^{n_Y}$, respectively. The X-program is a labeled directed acyclic graph with a designated start node, and each of whose nodes has outdegree 0 or 2. Each node of outdegree 2 is labeled either by an index in $\{1, \dots, n_X\}$ or by **receive**, and its two emanating edges are labeled 0 and 1, respectively. In addition to its 0 or 1 label, an edge may be labeled either by an output statement of the form $z_j = 0$ or $z_j = 1$ or a communication statement of the form **send**(0) or **send**(1). The Y-program is defined analogously.

The pair of branching programs computes a function $f : X \times Y \to Z \subseteq \{0, 1\}^{\leq n_Z}$ in the following natural way. Each program accesses its portion of the input and, starting at its start node, operates like a conventional branching program by following the edge labeled x_i (respectively, y_i) when encountering a node labeled i. Outputs are produced according to the output label on this edge, if any. When a program encounters a **receive** node it waits until the other program traverses an edge labeled **send**(b), and then the receiving program follows the edge labeled b. Similarly a program executing a **send** is blocked until the other program reaches a **receive** node. When a program reaches a node of outdegree 0, it halts. We require that each output bit may only be produced once on any given input pair. The function f is computed correctly on inputs x and y if the union of the outputs produced by the two programs comprises the bits of $f(x, y)$.

The *space* of each branching program is the base 2 logarithm of the number of its nodes. (This is the standard definition for branching programs [8], motivated by the fact that each node represents a different configuration of the program.) The *space* of the pair of programs is the maximum of the space of the two branching programs, and the *communication* is the length of the longest sequence of **send-receive** pairs executed on any input (x, y). The definitions can be generalized to communicating R-way branching programs for any R [7].

This model is a very natural one and a very general one as well. It can simulate, for example, two communicating space-bounded random access machines with a common write-only area for their output values.

One aspect of communicating branching programs that is somewhat subtle is the way in which output values are produced. Since all branchings of one of the programs that do not affect its communication with the other program are hidden from that other program, output values may be produced by one branching program without the explicit knowledge of the other branching program. In fact, all the bits communicated by the pair of branching programs may not be sufficient to determine the value of the function. However, the model in which all output values are communicated explicitly is a useful special case. We say that a pair of communicating branching programs is *open* if and only if the natural encoding of each output statement produced by either processor is communicated bit by bit to the other processor.

3. The general lower bound. The technique we develop here is an extension of the technique of Borodin et al. [7], [8] for time-space tradeoffs on sequential branching programs. Being purposely vague for the moment, their technique requires the following:

(1) a probability distribution on the set of inputs such that, with high probability, a large number of output bits are produced on any given input, and

(2) a proof that, given the distribution in (1), for any way of fixing a limited number of input variables, the probability that an input whose variables are so fixed produces a given set of k output bits whose values are fixed in any given way is exponentially small in k.

We develop a similar pair of conditions that allow proofs of communication-space tradeoffs. We first state our general technique for open pairs of communicating branching programs since this is the most natural argument. Then we outline how it can be extended to arbitrary pairs of communicating branching programs.

In order to motivate the properties that are appropriate for showing lower bounds for pairs of communicating branching programs, we first develop some facts about their operation.

Fix some $c > 0$ and any pair (u, v) of nodes in the pair of communicating branching programs, u in the X-program and v in the Y-program, and consider the action of the branching programs on input pair (x, y) starting at (u, v). For each input pair (x, y) we can follow the paths that the computation would take starting at (u, v) and stop when either a total of c bits of communication have been sent in both directions or the programs halt. (A third possibility is that there is no consistent computation on input (x, y) starting at (u, v), but any input (x, y) that reaches (u, v) will have a consistent computation. In the following definitions, we consider such an input pair (x, y) for which there is a consistent computation.) This produces a string of up to c communication bits, with fewer than c bits only if the programs halt before c bits of communication have been sent. Let $\gamma^c_{(u,v)}(x, y)$ be the following representation of this sequence of up to c bits communicated on input (x, y) starting at (u, v): represent each communicated bit b by two bits, using the extra bit to indicate which program sent b. For each string $\alpha \in \{0, 1\}^{\le 2c}$ we can define a set

$$R^\alpha_{(u,v)} = \{(x, y) \in X \times Y \mid \gamma^c_{(u,v)}(x, y) = \alpha\}.$$

A set $R \subseteq X \times Y$ is a *rectangle* if and only if there are sets $A \subseteq X$ and $B \subseteq Y$ such that $R = A \times B$.

LEMMA 3.1. *Let (u, v) be a pair of nodes in a pair of communicating branching programs, u in the X-program and v in the Y-program. The elements of $\{R^\alpha_{(u,v)} \mid \alpha \in \{0, 1\}^{\le 2c}\}$ are disjoint rectangles in $X \times Y$ whose union contains all input pairs (x, y) that reach (u, v).*

Proof. The fact that the sets $R^\alpha_{(u,v)}$ are disjoint is immediate from their definition. It is also clear that if (x, y) reaches (u, v), then $\gamma^c_{(u,v)}(x, y) = \alpha$ is defined and so $(x, y) \in R^\alpha_{(u,v)}$. The fact that each $R^{\alpha}_{(u,v)}$ is a rectangle follows by standard arguments in communication complexity (Yao [24]). It is proved inductively on the prefixes of α. □

We are now ready to state properties of a function that make it possible to prove communication-space tradeoffs. These properties will depend on certain parameters p, m, β, q, a, and K that will be set in later applications.

If $f(x, y) = z$, we will call the bits of x and y *input values* and the bits of z *output values*. For a function $f : X \times Y \to Z$ and a distribution $\mathcal{D}$ on $X \times Y$, the two properties are as follows:

Property A. There are $0 < p \leq 1$ and a positive integer m such that

$$\Pr_{\mathcal{D}}[f(x, y) \text{ has at least } m \text{ output values }] \geq p.$$

(Recall that, as (x, y) varies, $f(x, y)$ may have varying lengths.)

Property B. There are $0 < \beta < 1$, $0 < q < 1$, $a \geq 2$, and a positive integer K such that, for all positive integers $k \leq K$, the following holds: Let $R \subseteq X \times Y$ be any rectangle such that $\Pr_{\mathcal{D}}[(x, y) \in R] \geq q$. Then, for any set $V = \{z_{i_1} = b_1, \dots, z_{i_k} = b_k\}$ of k output values,

$$\Pr_{\mathcal{D}}[f(x, y) \text{ is consistent with } V \mid (x, y) \in R] \leq a\beta^k.$$

THEOREM 3.2. *Suppose that for* $f : X \times Y \to Z$ *there is a distribution* $\mathcal{D}$ *on* $X \times Y$ *such that Properties* A *and* B *hold with* $p > \max(a2^{-S+1}, 2a\beta^m)$. *Then any open pair* $\mathcal{P}$ *of communicating branching programs computing* f *using space* S *and communication* C *must satisfy*

$$C \cdot S = \Omega(m \log_a(1/\beta) \min(K, \log(p/q))).$$

(Note that, although the hypothesis $p > a2^{-S+1}$ refers to the space bound, it is even weaker than the relatively innocuous assumption $p > 2a/n$, where n is the number of input bits, since reading this many bits requires $S \geq \log_2 n$. Note also that, since there are 2^k choices for k output values and Property B must hold for all choices of output values, $2^k a\beta^k \geq 1$. Thus, $\log_a(1/\beta) \leq \log_a 2 + 1/k \leq 2$, so $\log_a(1/\beta)$ contributes at most a constant factor to the lower bound. However, β may be close to 1, so the $\log_a(1/\beta)$ factor in the lower bound may be close to 0.)

Proof. Let $\mathcal{P}$ be an open pair of communicating branching programs computing f, and let C and S be the communication and space, respectively, used by $\mathcal{P}$.

Case 1 ($S + 2 \geq \log_2(p/q)/4$). As explained above, $\log_a(1/\beta) \leq 2$. By Property A and the fact that $\mathcal{P}$ is open, $C \geq m$. Thus,

$$6CS \geq 2C(S+2) \geq m \log_a(1/\beta) \log_2(p/q)/4.$$

Case 2 ($S + 2 < \log_2(p/q)/4$). Let $c = \min(C, \lfloor \frac{1}{2} \log_2(p/q) - S - 1 \rfloor) \geq \min(C, \lfloor 2S + 4 - S - 1 \rfloor) > 0$. Fix any pair (u, v) of nodes, u in the X-program and v in the Y-program of $\mathcal{P}$. Fix $\alpha \in \{0, 1\}^{\leq 2c}$ such that α determines $k \leq K$ output values, that is, α contains the encoding of k output values. Let (x, y) be chosen at random according to $\mathcal{D}$. Suppose that $\Pr_{\mathcal{D}}[(x, y) \in R^{\alpha}_{(u,v)}] \geq q$. By Property B, Lemma 3.1, and the fact that $\mathcal{P}$ correctly computes f,

$$\Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \mid (x, y) \in R^{\alpha}_{(u,v)}] \leq a\beta^k. \tag{1}$$

Call $R^{\alpha}_{(u,v)}$ *tiny* if and only if $\Pr_{\mathcal{D}}[(x, y) \in R^{\alpha}_{(u,v)}] < q$. Let $T_{(u,v)}$ be the set of α such that $R^{\alpha}_{(u,v)}$ is tiny. For fixed (u, v) and varying α, Lemma 3.1 says that the sets $R^{\alpha}_{(u,v)}$ are disjoint and their union contains all pairs (x, y) that reach (u, v). Let

$$W = \{\alpha \in \{0, 1\}^{\leq 2c} \mid \alpha \text{ determines } k \text{ output values}\}.$$

For all (u, v) and all $k \leq K$,

$$\begin{aligned}
&\Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \ \wedge\ \gamma^{c}_{(u,v)}(x, y) \text{ determines } k \text{ output values}]\\
&\quad = \sum_{\alpha\in W} \Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \ \wedge\ (x, y) \in R^{\alpha}_{(u,v)}]\\
&\quad = \sum_{\alpha\in W\setminus T_{(u,v)}} \Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \ \wedge\ (x, y) \in R^{\alpha}_{(u,v)}]\\
&\qquad + \sum_{\alpha\in T_{(u,v)}} \Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \ \wedge\ (x, y) \in R^{\alpha}_{(u,v)}]\\
&\quad < \sum_{\alpha\in W\setminus T_{(u,v)}} \Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \ \wedge\ (x, y) \in R^{\alpha}_{(u,v)}] + |T_{(u,v)}|q\\
&\quad \leq 2^{2c+1}q + \sum_{\alpha\in W\setminus T_{(u,v)}} \Pr_{\mathcal{D}}[(x, y) \text{ reaches } (u, v) \mid (x, y) \in R^{\alpha}_{(u,v)}]\\
&\qquad\qquad \times \Pr_{\mathcal{D}}[(x, y) \in R^{\alpha}_{(u,v)}]\\
&\quad \leq 2^{2c+1}q + a\beta^{k} \sum_{\alpha\in W\setminus T_{(u,v)}} \Pr_{\mathcal{D}}[(x, y) \in R^{\alpha}_{(u,v)}]\\
&\quad \leq a\beta^{k} + 2^{2c+1}q,
\end{aligned} \tag{2}$$

the last two lines following from inequality (1) and the fact that the sets $R^{\alpha}_{(u,v)}$ are disjoint, respectively.

Inequality (2) is used in two different ways. For the first, by applying it to the start nodes u and v of their respective branching programs, we have

$$C > \min\left(K, \left\lfloor \frac{1}{2}\log_2(p/q) - S - 1\right\rfloor\right). \tag{3}$$

For assume to the contrary that $C \leq K$ and $C \leq \left\lfloor \frac{1}{2}\log_2(p/q) - S - 1\right\rfloor$. Then by the definition of c, $c = C$. Now choose $k = m$. By Property A and the fact that $\mathcal{P}$ is open, $k = m \leq C \leq K$, so inequality (2) holds. By the definition of c, $2^{2c+1}q \leq 2^{\log_2(p/q)-2+1}q = p/2$, so inequality (2) and Property A together yield $p \leq a\beta^{m} + 2^{2c+1}q \leq a\beta^{m} + p/2$. That is, $p/2 \leq a\beta^{m}$, which contradicts the hypothesis $p > 2a\beta^{m}$.

Now let $k = \min(K, m/\lceil C/c\rceil - 1)$ in inequality (2). Suppose $\mathcal{P}$ uses exactly C' bits of communication on input (x, y). Divide this sequence of C' bits into $\lceil C'/c\rceil$ segments, each consisting of c consecutive bits of communication (except possibly the last segment). If $f(x, y)$ has at least m output values, then (x, y) reaches some pair (u, v) of nodes such that $\gamma^{c}_{(u,v)}(x, y)$ determines at least $m/\lceil C'/c\rceil - 1 \geq m/\lceil C/c\rceil - 1 \geq k$ output values, for otherwise fewer than m output values are produced by $\mathcal{P}$ on input (x, y). (The -1 term accounts for the possibility that an output crosses a segment boundary.) Therefore, since $k \leq K$ and there are at most 2^{2S} node pairs (u, v),

$$\begin{aligned}
p &\leq \Pr_{\mathcal{D}}[(\exists u, v)((x, y) \text{ reaches } (u, v) \ \wedge\\
&\qquad\qquad \gamma^{c}_{(u,v)}(x, y) \text{ determines } k \text{ output values})]\\
&\leq 2^{2S}(a\beta^{k} + 2^{2c+1}q) = 2^{2S}a\beta^{k} + 2^{2S}2^{2c+1}q \leq 2^{2S}a\beta^{k} + p/2,
\end{aligned}$$

by the definition of c. Solving this yields $2S + \log_2(2a/p) \geq k\log_2(1/\beta)$. Since $p \geq a2^{-S+1}$, it follows that $3S \geq k\log_2(1/\beta)$.

Case 2.1 ($K < m/\lceil C/c\rceil - 1$). Then $k = K$. By Property A and the fact that $\mathcal{P}$ is open, $C \geq m$, so $3CS \geq mK\log_2(1/\beta) \geq mK\log_a(1/\beta)$.

Case 2.2 ($K \geq m/\lceil C/c\rceil - 1$). Then $k + 1 = m/\lceil C/c\rceil \geq mc/(C + c)$, so

$$
\begin{aligned}
12CS &\geq 6(C+c)S \\
&\geq 2(C+c)k\log_2(1/\beta) \\
&\geq (C+c)(k+1)\log_2(1/\beta) \\
&\geq mc\log_2(1/\beta) \\
&\geq m\log_2(1/\beta)\min\left(C, \left\lfloor \frac{1}{2}\log_2(p/q) - S - 1\right\rfloor\right) \\
&\geq m\log_2(1/\beta)\min\left(K, \left\lfloor \frac{1}{2}\log_2(p/q) - S - 1\right\rfloor\right) \qquad (4)\\
&\geq m\log_2(1/\beta)\min(K, \log_2(p/q)/4) \qquad (5)\\
&\geq m\log_a(1/\beta)\min(K, \log_2(p/q)/4).
\end{aligned}
$$

Inequality (4) follows from inequality (3), and inequality (5) from the condition of Case 2. □

Theorem 3.3 extends Theorem 3.2 to the case when the pair of communicating branching programs is not necessarily open.

THEOREM 3.3. *Suppose that for $f : X \times Y \to Z$ there is a distribution $\mathcal{D}$ on $X \times Y$ such that Properties* A *and* B *hold with $p > \max(a2^{-S+1}, 2a\beta^m)$. Then any pair $\mathcal{P}$ of communicating branching programs computing f using space S and communication C must satisfy*

$$(C + m\log m)\cdot S = \Omega(m\log_a(1/\beta)\min(K, \log(p/q))).$$

Proof. For any pair $\mathcal{P}$ of communicating branching programs, let the *observable behavior* of $\mathcal{P}$ on input (x, y) be the sequence of communicated bits and up to the first m output values produced on (x, y) by $\mathcal{P}$. Because of the structure of branching program pairs, there is no ambiguity about the order in which communication steps occur. However, the interleaving of the two programs' output values between communication steps is not determined, so for definiteness we assume that, between communication steps, outputs in the observable behavior produced by the X-program precede those produced by the Y-program. Since each output value is produced only once by the pair of branching programs, it is easy to see that, for any input pair, the observable behavior may be encoded using $O(C + m\log m)$ bits, where each output value is encoded using $O(\log m)$ bits that specify (1) that it is an output rather than a communication, (2) which output bit is being produced, (3) its value, and (4) which program produced it. Let $2C^*$ be the maximum, over all (x, y), of the length of the observable behavior on input (x, y). (We use $2C^*$ as the analogue of the $2C$ bits used in Theorem 3.2 to encode the communication in the strings $\gamma^c_{(u,v)}(x, y)$.)

The proof of Theorem 3.3 is analogous to that of Theorem 3.2, except that the communication C is replaced by the number of bits needed to describe the observable behavior, C^*. The main technical difference is in the definition of $\gamma^c_{(u,v)}(x, y)$, which, instead of being the string of length $2c$ describing the next c bits of communication on (x, y) starting at (u, v), is now the string of the next $2c$ bits of the observable behavior of $\mathcal{P}$ on (x, y) starting at (u, v). Since outputs are included in the observable behavior, the string $\gamma^c_{(u,v)}(x, y)$ determines output values just as the communication did in the case of an open pair of branching programs. The crucial fact, which is easily verified, is that the new $R^\alpha_{(u,v)}$ based on this definition of $\gamma^c_{(u,v)}(x, y)$ still are disjoint rectangles that cover the set of input pairs that arrive at (u, v).

The remainder of the proof is identical to that of Theorem 3.2 except for a couple of points. The part of the proof of Theorem 3.2 where the communication is divided into segments of length c is slightly different. When c bits of communication are replaced by $2c$ bits of observable behavior, it is no longer obvious that every boundary between segments of the computation on (x, y) can be chosen to be at a pair of nodes (u, v) since an output value may be produced along an edge and this is $O(\log m)$ bits of observable behavior as opposed to the single bit of communication that may occur on an edge. However, because the argument ignores any output whose production overlaps the boundary, if a boundary would naturally fall in the middle of an edge, then the boundary may be shifted past the edge to the subsequent node with no loss in the argument. The remaining difference is that, in the places where Property A and openness were used to show that $m \leq C$, in the modified proof $m \leq C^*$ follows directly from Property A and the definition of C^*. The conclusion of the argument is exactly the same with C^* replacing C as required. □

It is not too hard to see how the argument and Properties A and B can be modified to deal with R-way branching programs (Borodin and Cook [7]) or when the output values described in Properties A and B are of a restricted type (as in, for example, Abrahamson [1]).

4. Hash functions. We now apply the lower bound technique of the previous section to universal families of hash functions (Carter and Wegman [9], [10]). This will allow us to obtain lower bounds for a variety of interesting computational problems. We make use of a beautiful analog due to Mansour, Nisan, and Tiwari [17] of a lemma of Lindsey [4], [11] concerning Hadamard matrices.

Our results (and those in [17]) use the more restrictive definition of a universal family of hash functions given by Carter and Wegman in [10] (which they called "strongly universal" in [10]) rather than the somewhat broader definition given in [9]. To emphasize the nature of this stronger requirement we will call such families *pairwise universal.*

A pairwise universal family H of hash functions from a set X to a set Z satisfies the following two properties for h chosen uniformly at random from H:

1. For any $x \in X$, $h(x)$ is uniformly distributed in Z.
2. For any $x, x' \in X$ with $x \neq x'$ and for any $z, z' \in Z$, the events $h(x) = z$ and $h(x') = z'$ are independent.

We say that a pair of communicating branching programs computes the universal family of hash functions H if and only if it computes the function $f : X \times H \to Z$ given by $f(x, h) = h(x)$.

Of the two properties of a function required to apply our lower bound technique, Property B is the more difficult to prove. The following lemma on pairwise universal hash functions is critical in proving Property B for families of hash functions.

LEMMA 4.1 (Mansour, Nisan, and Tiwari [17]). *Let H be a pairwise universal family of hash functions from X to Z. Let $A \subseteq X$, $B \subseteq H$, and $E \subseteq Z$. Then*

$$\left| \Pr_{x\in A, h\in B}[h(x) \in E] - \frac{|E|}{|Z|} \right| \leq \sqrt{\frac{|H| \cdot |E|}{|A| \cdot |B| \cdot |Z|}}.$$

This lemma is used by Mansour, Nisan, and Tiwari [17] to prove time-space tradeoffs for computing hash functions. A somewhat weaker form of this lemma was proved independently by Yan [22] for the special case when the family of hash functions is given by matrix-vector product over GF(2).

THEOREM 4.2. *Let $\mathcal{P}$ be an open pair of communicating branching programs computing a pairwise universal family of hash functions from X to Z using communication C and space S. Let $n = \lfloor \log_2 |X| \rfloor$, $m = \lfloor \log_2 |Z| \rfloor - 1$, and $Z \subseteq \{0, 1\}^{\leq m+l}$, where $l < \min(\log_2 n, m) - 3$ is included to allow some slack in the output encoding. Then $C \cdot S = \Omega(nm/l)$.*

Proof. Let $\mathcal{D}$ be the uniform distribution on pairs (x, h). Since $h(x)$ is uniformly distributed in Z, Property A is satisfied with $p = 1/2$. Let $R = A \times B$ satisfy $|R| \geq 2^{K-n}|X \times H|$, where $K = n/2$. For any set $V = \{z_{i_1} = b_1, \ldots, z_{i_k} = b_k\}$ of $k \leq K$ output values, let

$E \subseteq Z$ be the set of vectors consistent with V. At most $2^{m+l-k+1}$ vectors are in E so that $|E|/|Z| \leq 2^{-(k-l)}$. Then Lemma 4.1 states that

$$\begin{aligned} \Pr_{\mathcal{D}}[h(x) \text{ is consistent with } V \mid (x,h) \in R] &\leq \frac{|E|}{|Z|} + \sqrt{\frac{|H|}{|R|} \cdot \frac{|E|}{|Z|}} \\ &\leq 2^{-(k-l)} + 1/\sqrt{2^{K-n}|X|2^{k-l}} \\ &\leq 2^{l+1} \cdot 2^{-k}. \end{aligned}$$

Thus Property B is satisfied with $q = 2^{-n/2}$, $\beta = 1/2$, $a = 2^{l+1}$, and $K = n/2$. Since $l < \min(\log_2 n, m) - 3$, $p = 1/2 > 2a/n$ and $p > 2a\beta^m$, so Theorem 3.2 implies that $C \cdot S = \Omega(nm/l)$. □

Theorem 4.3 extends Theorem 4.2 to the case in which the pair of communicating branching programs is not necessarily open.

THEOREM 4.3. *Any pair of communicating branching programs computing a pairwise universal family of hash functions from X to Z with communication C and space $S = o(n/(l \log m))$ satisfies $C \cdot S = \Omega(nm/l)$, where $n = \lfloor \log_2 |X| \rfloor$, $m = \lfloor \log_2 |Z| \rfloor - 1$, and $Z \subseteq \{0,1\}^{\leq m+l}$ for $l < \min(\log_2 n, m) - 3$.*

Proof. Using Theorem 3.3 in place of Theorem 3.2 in the proof of Theorem 4.2,

$$(C + m \log m)S = \Omega(nm/l).$$

Since, by hypothesis, $Sm \log m = o(nm/l)$, the conclusion $CS = \Omega(nm/l)$ follows. □

Similar statements to Theorem 4.3 can be made for each of the following corollaries. We simply state our results for open pairs of branching programs for convenience.

In the following corollaries, we can always choose l to be a constant.

COROLLARY 4.4. *Any open pair of communicating branching programs computing the product of an $n \times n$ matrix and an n-vector over* GF(2) *requires communication C and space S such that $C \cdot S = \Omega(n^2)$.*

COROLLARY 4.5. *If $r \geq 2^n$, then any open pair of communicating branching programs computing f :* $\mathrm{GF}(r) \times \mathrm{GF}(r)^2 \to \mathrm{GF}(r)$ *given by $f(x,(a,b)) = a \cdot x + b$ (in* GF(r)) *requires communication C and space S such that $C \cdot S = \Omega(n^2)$.*

The next two corollaries follow from Theorem 4.2 exactly as shown by Mansour, Nisan, and Tiwari [17] for time-space tradeoffs.

COROLLARY 4.6. *Any open pair of communicating branching programs computing the m-bit convolution of an n-bit string with an $(n + m - 1)$-bit string requires communication C and space S such that $C \cdot S = \Omega(nm)$.*

COROLLARY 4.7. *Any open pair of communicating branching programs computing the product of two $n \times n$ matrices over* GF(2) *requires communication C and space S such that $C \cdot S = \Omega(n^3)$.*

Corollaries 4.5, 4.6, and 4.7 are interesting in their own right and because they demonstrate tradeoffs in cases where the lower bound is greater than the total number of inputs that the two programs receive.

Using the natural generalization of communicating branching programs to pairs of r-way branching programs that are allowed to send and receive values in GF(r) one can prove, either by direct simulation or an analog of Theorem 3.2, the following analog of Theorem 4.2 for hash functions whose domain and range are vectors over GF(r).

THEOREM 4.8. *Any open pair of communicating r-way branching programs computing a pairwise universal family of hash functions from X to Z requires communication C and space S such that $C \cdot S = \Omega(nm(\log r)/l)$, where $n = \lfloor \log_r |X| \rfloor$, $m = \lfloor \log_r |Z| \rfloor - 1$, and $Z \subseteq (\mathrm{GF}(r))^{\leq m+l}$, for $l < \min(\log_r n, m) - 3$.*

This theorem has corollaries analogous to those of Theorem 4.2, such as the following.

COROLLARY 4.9. *Any open pair of communicating r-way branching programs computing the product of an $n \times n$ matrix and an n-vector over* GF(r) *requires communication C and space S such that $C \cdot S = \Omega(n^2 \log r)$.*

5. Open questions.

It is an interesting question whether or not similar bounds hold for $\wedge$-$\vee$ matrix-vector product. The results of Lam, Tiwari, and Tompa [15] show that such results do hold in a more restricted model in which the programs are restricted to being oblivious, i.e., straight-line, and the communication is one-way.

A natural approach to proving such a bound would be to try to prove Properties A and B for this problem using the distribution $\mathcal{D}$ employed by Babai, Frankl, and Simon [4] for proving a distributional communication complexity lower bound of $\Omega(\sqrt{n})$ for $\wedge$-$\vee$ dot product (i.e., set disjointness) and by Abrahamson [1] for proving a time-space tradeoff of $TS = \Omega(n^{1.5})$ on matrix-vector product. However, this approach cannot yield any interesting communication-space tradeoff since under this distribution, which chooses each input bit independently to be 1 with probability $1/\sqrt{n}$ and 0 with probability $(1 - 1/\sqrt{n})$, the program with the vector can simply communicate its value in expected $O(\sqrt{n} \log n)$ bits to the matrix program, which can store this value and perform the rest of the computation on its own.

An alternative approach would be to try to generalize the distribution on inputs that Razborov [20] used to prove that the distributional communication complexity of the set disjointness problem is $\Omega(n)$. Unfortunately, the fact that this distribution does not set the values of the inputs to the two programs independently creates serious problems when trying to generalize from a problem whose input consists of two vectors to a problem with a matrix and a vector as input. It seems unlikely that one can maintain sufficient independence between the inputs to the two programs while maintaining sufficient information content in the two inputs. It may be that the oblivious one-way result is leading us astray, but it seems more likely that we are unable to generalize it because our technique is fundamentally distributional in nature.

The question of the communication-space tradeoff for $\wedge$-$\vee$ matrix product and GF(2) matrix-vector product raises another interesting question. Suppose that function f on $X \times Y$ has ϵ-error distributional communication complexity (Yao [24], [26]) at least D_ϵ. Under what circumstances does the function F on $X^n \times Y$ given by $F((x_1, \ldots, x_n), y) = (f(x_1, y), \ldots, f(x_n, y))$ have communication-space tradeoff $\Omega(nD_\epsilon)$? As shown by Yao [24], [26] and extended by Babai, Frankl, and Simon [4], a lower bound $D_\epsilon \geq k$ can be obtained by showing that, for an appropriate distribution on $X \times Y$ under which f takes on each value at least a constant fraction of the time, any rectangle R, in which the probability of $f(x, y)$ taking on a particular value is less than ϵ, must have total probability at most $1/2^k$. This condition is very similar to our Property B, the important difference being that we require that this be true for ϵ much smaller than a constant, that is, for $\epsilon = \beta^k$ for $\beta < 1$. If the only rectangles $A \times B$ in $X^n \times Y$ had A of the form $A_1 \times \cdots \times A_n$, then there would be a direct translation of distributional communication complexity lower bounds for f to those for F. It is not clear what conditions on f will allow the handling of general A as well. The technique of Mansour, Nisan, and Tiwari [17] and Yan [22] implies that it is sufficient to have not only a small probability of a value in such a rectangle R but also a small variance in the probability of the value occurring in the rows (or columns) of R.

Acknowledgments. We thank Johan Håstad, Noam Nisan, Larry Ruzzo, and Prasoon Tiwari for helpful comments. We are particularly grateful to the anonymous referees for their very conscientious and constructive reviews.

REFERENCES

[1] K. Abrahamson, *A time-space tradeoff for boolean matrix multiplication*, in 31st Annual Symposium on Foundations of Computer Science, St. Louis, MO, Oct. 1990, IEEE, pp. 412–419.

[2] ———, *Time-space tradeoffs for algebraic problems on general sequential models*, J. Comput. System Sci., 43 (1991), pp. 269–289.

[3] A. V. Aho, J. D. Ullman, and M. Yannakakis, *On notions of information transfer in VLSI circuits*, in Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, Boston, MA, Apr. 1983, pp. 133–139.

[4] L. Babai, P. Frankl, and J. Simon, *Complexity classes in communication complexity theory*, in 27th Annual Symposium on Foundations of Computer Science, Toronto, Ontario, Oct. 1986, IEEE, pp. 337–347.

[5] L. Babai, N. Nisan, and M. Szegedy, *Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs*, J. Comput. System Sci., 45 (1992), pp. 204–232.

[6] P. Beame, *A general sequential time-space tradeoff for finding unique elements*, in Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing, Seattle, WA, May 1989, pp. 197–203.

[7] A. Borodin and S. A. Cook, *A time-space tradeoff for sorting on a general sequential model of computation*, SIAM J. Comput., 11 (1982), pp. 287–297.

[8] A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa, *A time-space tradeoff for sorting on non-oblivious machines*, J. Comput. System Sci., 22 (1981), pp. 351–364.

[9] J. L. Carter and M. N. Wegman, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[10] ———, *New hash functions and their use in authentication and set equality*, J. Comput. System Sci., 22 (1981), pp. 265–277.

[11] P. Erdös and J. Spencer, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[12] D. Y. Grigoriev, *An application of separability and independence notions for proving lower bounds of circuit complexity*, in Notes of Scientific Seminars 60, Steklov Mathematical Institute, Leningrad Department, 1976, pp. 38–48. (In Russian.)

[13] A. Hajnal, W. Maass, and G. Turán, *On the communication complexity of graph properties*, in Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, May 1988, pp. 186–191.

[14] M. Karchmer and A. Wigderson, *Monotone circuits for connectivity require super-logarithmic depth*, in Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, May 1988, pp. 539–550.

[15] T. W. Lam, P. Tiwari, and M. Tompa, *Trade-offs between communication and space*, J. Comput. System Sci., 45 (1992), pp. 296–315.

[16] R. J. Lipton and R. Sedgewick, *Lower bounds for VLSI*, in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, Milwaukee, WI, May 1981, pp. 300–307.

[17] Y. Mansour, N. Nisan, and P. Tiwari, *The computational complexity of universal hashing*, in Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing, Baltimore, MD, May 1990, pp. 235–243.

[18] K. Mehlhorn and E. M. Schmidt, *Las Vegas is better than determinism in VLSI and distributed computing*, in Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 1982, pp. 330–337.

[19] R. Raz and A. Wigderson, *Monotone circuits for matching require linear depth*, J. Assoc. Comput. Mach., 39 (1992), pp. 736–744.

[20] A. A. Razborov, *On the distributional complexity of disjointness*, in Automata, Languages, and Programming: 17th International Colloquium, Lecture Notes in Computer Science 443, Warwick University, England, July 1990, Springer-Verlag, New York, pp. 249–253.

[21] M. Tompa, *Time-space tradeoffs for computing functions, using connectivity properties of their circuits*, J. Comput. System Sci., 20 (1980), pp. 118–132.

[22] P. Yan, *A tradeoff between communication and space*, manuscript, 1989.

[23] M. Yannakakis, *Expressing combinatorial optimization problems by linear programs*, in Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, Chicago, IL, May 1988, pp. 223–228.

[24] A. C. Yao, *Some complexity questions related to distributive computing*, in Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, GA, April–May 1979, pp. 209–213.

[25] ———, *The entropic limitations of VLSI computations*, in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, Milwaukee, WI, May 1981, pp. 308–311.

[26] ———, *Lower bounds by probabilistic arguments*, in 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, Nov. 1983, IEEE, pp. 420–428.

[27] Y. Yesha, *Time-space tradeoffs for matrix multiplication and the discrete Fourier transform on any general sequential random-access computer*, J. Comput. System Sci., 29 (1984), pp. 183–197.

SIAM J. COMPUT.
Vol. 23, No. 3, pp. 662–669, June 1994

A NEW INSIGHT INTO THE COFFMAN–GRAHAM ALGORITHM*

BERTRAND BRASCHI† AND DENIS TRYSTRAM†

Abstract. The approximate solution of the m-machine problem is addressed. The Lam–Sethi worst-case analysis of the Coffman–Graham algorithm is set up to be partly incorrect. A slightly different context is set up to correct and complete this analysis. It is shown that the makespan of a schedule computed by an extended Coffman–Graham algorithm is lower than or at worst equal to $(2 - 2/m)\omega_{\text{opt}} - (m - 3)/m$, where ω_{opt} is the minimal makespan of a schedule.

Key words. scheduling theory, worst-case analysis, list scheduling

AMS subject classification. 90B35

1. Introduction. The m-machine problem is to schedule a set of tasks on a given number of identical machines, m, with a minimum makespan. The tasks are subject to precedence constraints and have the same unit execution time. Preemption is not allowed. The Coffman–Graham algorithm (CG-algorithm) is the only polynomial algorithm which solves exactly the two-machine problem [1], [3], [4] and whose worst-case behavior has been analyzed for more than two machines.

We present the m-machine problem in §2 and we give the elements of the Coffman–Graham analysis in §3. We show in §4 that a part of the worst-case analysis of the CG-algorithm published by Lam and Sethi [6], for two machines or more, is not correct. In §5, an extension of the CG-algorithm is considered. We correct the contentious point of the Lam–Sethi analysis and refine their worst-case bound for this extended algorithm in §6.

2. Definitions. The set of tasks and the precedence relation among the tasks are represented by an acyclic directed graph $(\mathcal{C}, \ll)$ (see Fig. 1). Let T and T' be two tasks of $\mathcal{C}$; we note $T \ll T'$ if and only if (iff) (T, T') belongs to $\ll$. T is a *predecessor* of T', and T' a *successor* of T iff $T \ll T'$. The precedence relation between T and T' is *immediate* iff there exists no task T'' such that $T \ll T''$ and $T'' \ll T'$. The set of immediate successors of T is noted $\mathcal{S}(T)$. A set of tasks C is a *chain* iff the restriction of the transitive closure of $(\mathcal{C}, \ll)$ to C is a total order on C. Let $\mathcal{S}$ and $\mathcal{S}'$ be two sets of tasks. We say that $\mathcal{S}$ precedes $\mathcal{S}'$ and note $\mathcal{S} \ll \mathcal{S}'$ iff each task of $\mathcal{S}$ precedes all the tasks of $\mathcal{S}'$. The concept of chain applies also to a collection of sets of tasks. A schedule of $(\mathcal{C}, \ll)$ on a set of m machines $[M_1, \dots, M_m]$ is defined by assigning to every task T a machine $\mu(T)$ and a *slot* of execution $\lambda(T)$, a positive integer, such that the following conditions are verified:

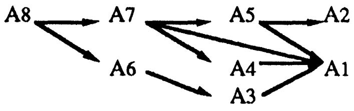
- if $T \ll T'$, then $\lambda(T) < \lambda(T')$—a task is executed before its possible successors,
- if $\mu(T) = \mu(T')$, then $\lambda(T) \neq \lambda(T')$—a machine executes only one task during a slot.

The definition of λ is extended to the sets of tasks: $\lambda(\mathcal{S}) = \{\lambda(T)/T \in \mathcal{S}\}$. We note $\omega(\mathcal{S})$ the makespan of $\mathcal{S}$: $\omega(\mathcal{S}) = |\lambda(\mathcal{S})|$, and we define Idle$(\mathcal{S})$ as $m\omega(\mathcal{S}) - |\mathcal{S}|$. The schedule of $(\mathcal{C}, \ll)$ is said to be *optimal* iff $\omega(\mathcal{C})$ is minimal. The minimal makespan is noted $\omega_{\text{opt}}(\mathcal{C})$. We define the optimal makespan of a subset $\mathcal{S}$ of $\mathcal{C}$, $\omega_{\text{opt}}(\mathcal{S})$, as the makespan of the optimal schedule for the restriction of the transitive closure of $(\mathcal{C}, \ll)$ to $\mathcal{S}$: We note: $\text{Idle}_{\text{opt}}(\mathcal{S}) = m\omega_{\text{opt}}(\mathcal{S}) - |\mathcal{S}|$.

3. The Coffman–Graham analysis [2]. The CG-algorithm begins to assign to each task T a positive integer $\alpha(T)$, its *priority*. These priorities follow a rule, which we translate into

*Received by the editors May 21, 1990; accepted for publication (in revised form) October 16, 1992.

†Laboratoire de Modélisation et de Calcul, I.M.A.G., 46 Av. F. Viallet, 38031 Grenoble, France.



	slot 1	slot 2	slot 3	slot 4	slot 5
M1	A 8	A 7	A 5	A 3	A 1
M2		A 6	A 4	A 2	

FIG. 1. *A precedence graph* $(\mathcal{C}, \ll)$ *and one of its possible schedules on two machines. The tasks of* $\mathcal{S} = \{A_2, A_1\}$ *are executed during the slots of* $\lambda(\mathcal{S}) = \{4, 5\}$: $\omega(\mathcal{S}) = 2$ *and* $\text{Idle}(\mathcal{S}) = 2$. *There is no precedence relation between* A_1 *and* A_2 : A_1 *and* A_2 *could be executed during the same slot. Thus the optimal makespan of* $\mathcal{S}$ *is* $\omega_{\text{opt}}(\mathcal{S}) = 1$ *and* $\text{Idle}_{\text{opt}}(\mathcal{S}) = 0$.

the *CG-rule* below, where $\alpha(\mathcal{S})$ denotes the set of priorities for the tasks set $\mathcal{S}$ and $\alpha(\emptyset)$ is equal to $\{0\}$.

CG-RULE. *If* $T \neq T'$, *then* $\alpha(T) \neq \alpha(T')$, *and if* $\alpha(T) \geq \alpha(T')$, *then* $\max(\alpha(\mathcal{S}(T)\backslash\mathcal{S}(T'))) \geq \max(\alpha(\mathcal{S}(T')\backslash\mathcal{S}(T)))$.

Then the CG-algorithm computes a schedule such that if there exists a task T and a slot L lower than $\lambda(T)$ during which a machine is idle or executes a task whose priority is lower than $\alpha(T)$, then T has a predecessor executed during L. Such a schedule is said to be a *list schedule* [5].

We consider a schedule computed by the CG-algorithm and assume that the tasks are executed on $M_1, M_2, \ldots$, in decreasing order of priority during a given slot. We note $\text{Next}(T)$ the task executed on M_1 during the slot $\lambda(T) + 1$, if such a task exists.

A series of *critical tasks* is defined by a right to left scanning of the schedule (see Fig. 2). Let us assume that the critical task U_i, $i \geq 0$, is defined. If there exists at least one slot lower than $\lambda(U_i)$, during which the task executed by M_2 has a priority lower than $\alpha(U_i)$, then the critical task U_{i+1} is defined to be the task executed by M_1 during the rightmost of these slots. A *block* $\mathcal{X}_i$ is then defined so that U_{i+1} precedes each task of $\mathcal{X}_i$:

$$\mathcal{X}_i = \{T/\alpha(T) \geq \alpha(U_i) \quad \text{and} \quad \lambda(T) > \lambda(U_{i+1})\}.$$

If U_{i+1} cannot be defined then $\mathcal{X}_i$ is defined as:

$$\mathcal{X}_i = \{T/\alpha(T) \geq \alpha(U_i)\}.$$

The first critical task, U_0, is the task executed by M_1 during the rightmost slot of $\lambda(\mathcal{C})$. The following lemma is derived from the block definition:

BLOCK COMPOSITION LEMMA. *At least two tasks of a block* $\mathcal{X}_i$ *are executed during each slot of* $\lambda(\mathcal{X}_i)$ *except the last slot* $\lambda(U_i)$.

4. The Lam–Sethi analysis [6]. Lam and Sethi define a second layer of tasks sets upon the blocks layer, the *segments*. The segments can be characterized by the *segment composition lemma*:

SEGMENT COMPOSITION LEMMA. *A segment* $\mathcal{W}$ *is the disjoint union of* k *consecutive blocks,* $k \geq 1$, *and of a set of at least* $k - 1$ *additional tasks. Each additional task is preceded by a task of* $\mathcal{W}$.

Lam and Sethi bound $\omega(\mathcal{W})$ with respect to $\omega_{\text{opt}}(\mathcal{W})$ by means of a chain of tasks of $\mathcal{W}$. We show that each task of this chain does not necessarily belong to $\mathcal{W}$, as Lam and Sethi presume.

An inductive argument employing the CG-rule makes it clear that the priorities are compatible with the precedence relation: if $T \ll T'$, then $\alpha(T) > \alpha(T')$. The following property is a consequence of this compatibility (Lemma 2.1 of [6]):

MONOTONY PROPERTY. *If* T *is executed on* M_1 *and* $\lambda(T) \leq \lambda(T')$, *then* $\alpha(T) \geq \alpha(T')$.

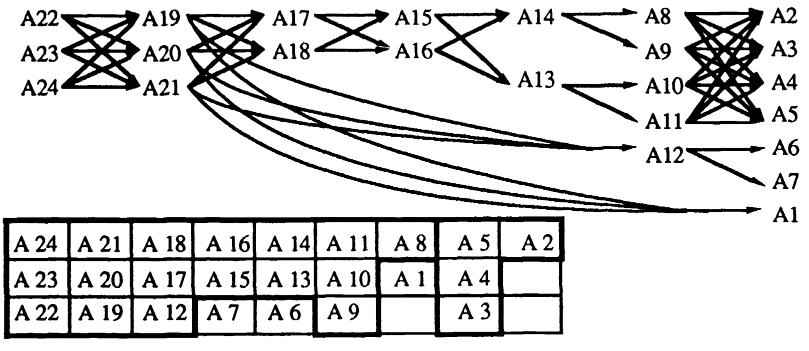
According to Lam and Sethi, a task executed during a slot of $\lambda(\mathcal{W})$ that does not belong to a block, is called an *extra task* and a slot of $\lambda(\mathcal{W})$ during which an extra task is executed is called a *partial slot*. The p partial slots of $\lambda(\mathcal{W})$ are noted $\lambda_1, \ldots, \lambda_p$ from left to right. The task executed by M_1 during λ_i is noted T_i. The following lemma relates the partial slots to the precedence constraints.

LAM–SETHI LEMMA. *Let λ_i be a partial slot of $\lambda(\mathcal{W})$ such that* $\mathrm{Next}(T_i)$ *exists. Every task R_i such that $\alpha(R_i) \geq \alpha(T_i)$ has an immediate successor R_{i+1} such that $\alpha(R_{i+1}) \geq \alpha(\mathrm{Next}(T_i))$.*

Proof. It can be induced from the monotony property and the block definition that any extra task E either is executed during the last slot of $\lambda(\mathcal{W})$ or has a priority lower than $\alpha(\mathrm{Next}(E))$.

Let E be an extra task executed during λ_i such that $\mathrm{Next}(T_i)$ exists. Obviously we have $\alpha(E) < \alpha(\mathrm{Next}(E))$, and because $\mathrm{Next}(T_i) = \mathrm{Next}(E)$, there exists an immediate predecessor T^* of $\mathrm{Next}(T_i)$, executed during λ_i. The task T_i is executed on M_1, and thus the inequality $\alpha(T_i) \geq \alpha(T^*)$ holds and therefore by transitivity, $\alpha(R_i) \geq \alpha(T^*)$. According to the CG-rule, an immediate successor R_{i+1} of R_i, such that $\alpha(R_{i+1}) \geq \alpha(\mathrm{Next}(T_i))$, must exist. □

If the slot λ_i is not the last partial slot of $\lambda(\mathcal{W})$, the monotony property makes it possible to state that $\alpha(\mathrm{Next}(T_i)) \geq \alpha(T_{i+1})$, thus, by transitivity, $\alpha(R_{i+1}) \geq \alpha(T_{i+1})$. This enables Lam and Sethi to conclude, by induction, that every task T such that $\alpha(T) \geq \alpha(T_i)$ precedes a chain of at least $p - i$ tasks. The point is that these tasks do not necessarily belong to $\mathcal{W}$. It is actually possible that $\alpha(R_i) < \alpha(U_j)$ even if $\alpha(R_i) \geq \alpha(\mathrm{Next}(T_i))$, where $\alpha(U_j)$ determines if R_i belongs to a block or not. Then, we cannot decide whether R_i belongs to $\mathcal{W}$ or not. Hence, the number of tasks in such a chain cannot be used as a lower bound for $\omega_{\mathrm{opt}}(\mathcal{W})$, as Lam and Sethi do. The definition of a segment given by Lam and Sethi is too narrow: in Fig. 2, the tasks A_6 and A_7 are similar but A_7 belongs to the segment, which is not the case for A_6. We modify the segment definition in §6 to add to the blocks of a segment as many tasks as possible.



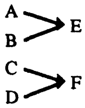
A 24	A 21	A 18	A 16	A 14	A 11	A 8	A 5	A 2
A 23	A 20	A 17	A 15	A 13	A 10	A 1	A 4	
A 22	A 19	A 12	A 7	A 6	A 9		A 3	

FIG. 2. *Analysis of the structure of a schedule on three machines. The tasks are given priorities according to the CG-rule. The priority of the task A_i is its index: $\alpha(A_i) = i$. The task U_0 is A_2. The first task executed on M_2 before $\lambda(U_0)$ with a priority lower than $\alpha(U_0)$ is A_1: the task U_1 is A_8 and $\mathcal{X}_0 = \{A_2, A_3, A_4, A_5\}$. There is no task executed on M_2 before $\lambda(U_1)$ with a priority lower than $\alpha(U_1)$: $\mathcal{X}_1 = \{A_8, A_9, A_{10}, A_{11}, A_{13}, A_{14}, A_{15}, A_{16}, A_{12}, A_{17}, A_{18}, \ldots, A_{24}\}$. The task A_{12} belongs to $\mathcal{X}_1$, has no successor in $\mathcal{X}_1$ and does not precede all the tasks of $\mathcal{X}_0$: A_{12} has at least one successor of priority greater than or equal to $\alpha(U_0)$ executed during a slot of $\lambda(\mathcal{X}_1)$. The task A_6, or A_7, is such a successor. A segment $\mathcal{W}$ can be set as the disjoint union of $\mathcal{X}_1$, $\{A_7\}$, and $\mathcal{X}_0$. There are four partial slots: T_1 is A_{16}, T_2 is A_{14}, T_3 is A_8, and T_4 is A_2. Since $\alpha(A_{24}) \geq \alpha(T_1)$, there exists a successor of A_{24}, say A_{21}, such that $\alpha(A_{21}) \geq \alpha(T_2)$. Recursively, $A_{21} \ll A_{12}(\alpha(A_{12}) \geq \alpha(T_3))$, and $A_{12} \ll A_6(\alpha(A_6) \geq \alpha(T_4))$. In conclusion, A_{24} precedes the chain $\{A_{21}, A_{12}, A_6\}$ but A_6 does not belong to $\mathcal{W}$.*

5. An extended algorithm. The CG-algorithm makes an arbitrary choice to assign different priorities to two different tasks T and T' such that $\max(\alpha(\mathcal{S}(T)\backslash\mathcal{S}(T')))$ is equal to $\max(\alpha(\mathcal{S}(T')\backslash\mathcal{S}(T)))$. This choice restricts the set of priorities the predecessors of T and T' can have, and thus, the set of schedules the CG-algorithm can compute. Let us consider a straightforward extension of the CG-algorithm, the *ECG-algorithm*, which assigns to each task T a priority $\alpha(T)$ according to the following *relaxed rule* and then computes a list schedule according to these priorities:

RELAXED RULE. *If* $\alpha(T) \geq \alpha(T')$, *then* $\max(\alpha(\mathcal{S}(T)\backslash\mathcal{S}(T'))) \geq \max(\alpha(\mathcal{S}(T')\backslash\mathcal{S}(T)))$.

Any schedule computed by the CG-algorithm can be computed by the ECG-algorithm but the converse is not true, as depicted in Fig. 3. We show in §6 that the main ideas of Lam and Sethi still hold for the ECG-algorithm.



Tasks	A	B	C	D	E	F
An instance of CG-priorities	6	5	4	3	2	1
An instance of relaxed priorities	2	2	2	2	1	1

A list schedule according to the instance of relaxed priorities	A	B	E
	C	D	F

FIG. 3. *Contribution of the ECG-algorithm. According to the CG-rule, either α (A) and $\alpha(B)$ are greater than $\alpha(C)$ and $\alpha(D)$ or the inverse is true. Thus, in any schedule computed by the CG-algorithm on two machines, A and B are executed during the same slot. On the other hand, we show a list schedule on two machines, based on priorities which follow the relaxed rule, where A and B are not executed during the same slot.*

6. An extended analysis. Now we consider a schedule computed by the ECG-algorithm. Let the critical tasks and the blocks be defined as in §3. Any critical task still precedes each task of the block to its right, if it exists. The block composition lemma and the monotony property remain valid. We give a new definition of the segments.

6.1. The segment definition. A series of *critical blocks* is defined by a left-to-right scanning of the schedule (see Fig. 4). Let us assume that the critical block $\mathcal{Y}_j$, $j \geq 0$, is defined. The set of tasks $\mathcal{W}_j(\mathcal{X}_i)$ is defined for each block $\mathcal{X}_i$ as follows:

$$\mathcal{W}_j(\mathcal{X}_i) = \{T/T \in \mathcal{Y}_j \text{ or } T \text{ is preceded by a task of } \mathcal{Y}_j \text{ and } \alpha(T) \geq \alpha(U_{i+1})\}.$$

If there exists at least one block $\mathcal{X}_i$ to the right of $\mathcal{Y}_j$ such that $\mathcal{W}_j(\mathcal{X}_i) \ll \mathcal{X}_i$, then $\mathcal{Y}_{j+1}$ is defined to be the leftmost of these blocks. The *segment* $\mathcal{W}_j$ is defined as $\mathcal{W}_j(\mathcal{Y}_{j+1})$. If there is no block $\mathcal{X}_i$ to the right of $\mathcal{Y}_j$ such that $\mathcal{W}_j(\mathcal{X}_i) \ll \mathcal{X}_i$, then $\mathcal{W}_j$ is defined as follows:

$$\mathcal{W}_j = \{T/T \in \mathcal{Y}_j \text{ or } T \text{ is preceded by a task of } \mathcal{Y}_i \text{ and } \alpha(T) \geq \alpha(U_0)\}.$$

The first critical block, $\mathcal{Y}_0$, is the leftmost block. We note ${}^jU$ and U^j, the leftmost and the rightmost critical tasks of $\mathcal{W}_j$, respectively.

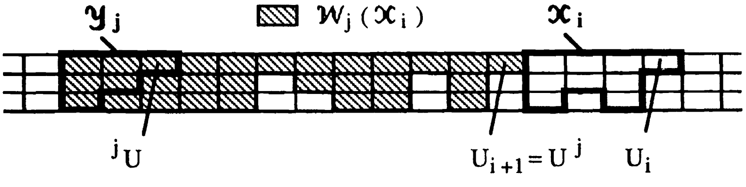


FIG. 4. *The segment definition. $\mathcal{W}_j$ is equal to $\mathcal{W}_j(\mathcal{X}_i)$, ${}^jU$ is equal to the critical task of $\mathcal{Y}_j$ and U^j is U_{i+1}.*

6.2. The segment composition lemma. We use the arguments of Lemmas 2.3 and 2.2 in [6] and show that the segment composition lemma holds in this context. Let us consider a critical block $\mathcal{Y}_j$ and a block $\mathcal{X}_i$ to the right of $\mathcal{Y}_j$.

If $\mathcal{W}_j(\mathcal{X}_i)$ does not precede $\mathcal{X}_i$, then there exists a task T of $\mathcal{W}_j(\mathcal{X}_i)$ and a task T' of $\mathcal{X}_i$ such that T does not precede T'. The tasks T and T' can be respectively considered without successor in $\mathcal{W}_j(\mathcal{X}_i)$ and without predecessor in $\mathcal{X}_i$. The critical task U_{i+1} precedes T' and thus T' belongs to $\mathcal{S}(U_{i+1})$. Furthermore, the priorities follow the relaxed rule and the fact that T belongs to $\mathcal{W}_j(\mathcal{X}_i)$ implies that $\alpha(T) \geq \alpha(U_{i+1})$. Hence, there exists a task B such that B belongs to $\mathcal{S}(T)\backslash\mathcal{S}(U_{i+1})$ and

$$(1) \qquad \alpha(B) \geq \max(\alpha(\mathcal{S}(U_{i+1})\backslash\mathcal{S}(T))).$$

Because B is not an immediate successor of U_{i+1}, if B were a successor of U_{i+1}, an immediate successor A of U_{i+1} preceding B should exist. As the priorities are compatible with the precedence relation, A should verify $\alpha(A) > \alpha(B)$ and thus, according to (1), A should be an immediate successor of T. This would contradict the fact that B is an immediate successor of T. Therefore, B is not a successor of U_{i+1}.

For T' belongs to $\mathcal{S}(U_{i+1})$, the inequality (1) induces that $\alpha(B) \geq \alpha(T')$, and for T' belongs to $\mathcal{X}_i$, $\alpha(B) \geq \alpha(U_i)$. Hence, B cannot be executed after $\lambda(U_{i+1})$ because otherwise it would belong to $\mathcal{X}_i$ and therefore be preceded by U_{i+1}. This means that B neither belongs to $\mathcal{W}_j(\mathcal{X}_i)$ nor to $\mathcal{X}_i$. In conclusion, the task B is a successor of T and thus is preceded by a task of $\mathcal{Y}_j$ and $\alpha(B) \geq \alpha(U_i)$. The disjoint union of $\mathcal{W}_j(\mathcal{X}_i)$, $\mathcal{X}_i$, and $\{B\}$ is included in $W_j(\mathcal{X}_{i-1})$, if $\mathcal{X}_{i-1}$ exists, or in $\mathcal{W}_j$, if $\mathcal{X}_{i-1}$ does not exist. This leads by induction to the segment composition lemma.

We point out that we have modified the scope of the set of *additional tasks*. No task T such that $\alpha(T) \geq \alpha(U^j)$ executed during $\lambda(\mathcal{W}_j)$ is excluded from $\mathcal{W}_j$, as it actually appears to be the problem on Fig. 2.

6.3. The partial-slot lemma. We continue the reasoning that has led to the Lam–Sethi lemma to build a chain and ensure that each task of the chain belongs to a given segment $\mathcal{W}_j$. We consider λ_i a partial slot of $\mathcal{W}_j$ that is not the rightmost, if such a slot exists.

If the task T_i is a critical task, then it is clear that $T_i \ll \text{Next}(T_i)$. If T_i is not a critical task, then because of the monotony property and the block definition, every extra task E executed during λ_i is such that $\alpha(E) < \alpha(\text{Next}(E))$. Thus, following the proof of the Lam–Sethi lemma, T_i has an immediate successor of priority greater than or equal to $\alpha(\text{Next}(T_i))$.

Therefore, if R_i is a task of $\mathcal{W}_j$ such that $\alpha(R_i) \geq \alpha(T_i)$, then R_i has an immediate successor R_{i+1} such that $\alpha(R_{i+1}) \geq \alpha(\text{Next}(T_i))$. Because of the monotony property, we can state that $\alpha(R_{i+1}) \geq \alpha(T_{i+1}) \geq \alpha(U^j)$. These inequalities mean that R_{i+1} belongs to $\mathcal{W}_j$ and provide the material for the straightforward recursive proof leading to the partial-slot lemma.

PARTIAL-SLOT LEMMA. *Let $\mathcal{W}$ be a segment and $\lambda_1, \ldots, \lambda_p$ be the p partial slots of $\lambda(\mathcal{W})$. Every task of $\mathcal{W}$ with a priority greater than or equal to $\alpha(T_i)$ precedes a chain of at least $p - i$ tasks of $\mathcal{W}$.*

6.4. The local-bound lemma. We consider a segment $\mathcal{W}_j$ and we bound the worst makespan $\omega(\mathcal{W}_j)$ the ECG-algorithm can compute. Let k be the number of blocks included in $\mathcal{W}_j$, $\mathcal{X}$ be the union of these blocks, $\mathcal{A}$ be the set $\mathcal{W}_j\backslash\mathcal{X}$ of additional tasks, ${}^jU$ and $\mathcal{Y}_j$ be the leftmost critical task and the leftmost critical block of $\mathcal{W}_j$, and $\lambda_1, \ldots, \lambda_p$ be the p partial slots of $\lambda(\mathcal{W}_j)$. We note $\mathcal{F}$ the set of the tasks of $\mathcal{X}$ executed during λ_1.

- We bound $\text{Idle}(\mathcal{W}_j)$ with respect to p and $\mathcal{F}$. It is easy to verify that $\lambda(\mathcal{W}_j)$ and $\lambda(\mathcal{X})$ are equal and that $\text{Idle}(\mathcal{W}_j)$ is equal to $\text{Idle}(\mathcal{X}) - |\mathcal{A}|$. The set $\lambda(\mathcal{W}_j)$ is divided into three disjoint subsets by checking the type of each slot:

$\mathcal{L}_1 = \{L \in \lambda(\mathcal{W}_j)/L$ is a partial slot during which only one task of $\mathcal{X}$ is executed$\}$,
$\mathcal{L}_{1,m} = \{L \in \lambda(\mathcal{W}_j)/L$ is a partial slot during which at least two tasks of $\mathcal{X}$ are executed$\}$,
$\mathcal{L}_m = \{L \in \lambda(\mathcal{W}_j)/L$ is not a partial slot$\}$.

The block composition lemma entails that

$$(2) \qquad |\mathcal{L}_1| \leq k \qquad (k \text{ is the number of blocks included in } \mathcal{W}_j),$$

and the partial-slot definition leads to

$$(3) \qquad |\mathcal{L}_1| + |\mathcal{L}_{1,m}| = p \qquad (p \text{ is the number of partial slots of } \lambda(\mathcal{W}_j)).$$

The definition of $\mathcal{L}_1$, $\mathcal{L}_{1,m}$, and $\mathcal{L}_m$ and the equality between Idle(W_j) and Idle($\mathcal{X}$) $- |\mathcal{A}|$ lead to

$$(4) \qquad \text{Idle}(\mathcal{W}_j) \leq (m-1)|\mathcal{L}_1| + (m-2)|\mathcal{L}_{1,m}| - |\mathcal{A}|.$$

We substitute $|\mathcal{L}_1|$ and $|\mathcal{L}_{1,m}|$ using (2) and (3) and substitute $|\mathcal{A}|$ by $k-1$, according to the segment composition lemma:

$$(5) \qquad \text{Idle}(\mathcal{W}_j) \leq (p+1)(m-2) - (m-3).$$

Furthermore, if the first slot of $\lambda(\mathcal{W}_j)$ is partial, we derive the inequality (4′), if λ_1 belongs to $\mathcal{L}_1$ or (4″), if λ_1 belongs to $\mathcal{L}_{1,m}$, instead of (4):

$$(4') \qquad \text{Idle}(\mathcal{W}_j) \leq \text{Idle}(\mathcal{F}) + (m-1)(|\mathcal{L}_1| - 1) + (m-2)|\mathcal{L}_{1,m}| - |\mathcal{A}|,$$

$$(4'') \qquad \text{Idle}(\mathcal{W}_j) \leq (m-1)|\mathcal{L}_1| + \text{Idle}(\mathcal{F}) + (m-2)(|\mathcal{L}_{1,m}| - 1) - |\mathcal{A}|.$$

In either case, we can derive

$$(5') \qquad \text{Idle}(\mathcal{W}_j) \leq \text{Idle}(\mathcal{F}) + p(m-2) - (m-3).$$

- We now establish lower bounds for $\omega_{\text{opt}}(\mathcal{W}_j)$ and $\text{Idle}_{\text{opt}}(\mathcal{W}_j)$ with respect to p and $\mathcal{F}$. It can be induced from the partial-slot lemma that each task T such that $\alpha(T) \geq \alpha(T_1)$ precedes a chain of at least $p-1$ tasks of $\mathcal{W}_j$.

Let us assume that the first slot of $\lambda(\mathcal{W}_j)$ is not partial. If a task T of $\mathcal{W}_j$ is such that $\lambda(T) < \lambda(T_1)$ and $\alpha(T) < \alpha(T_1)$, then, because the ECG-algorithm computes a list schedule, at least one task of $\mathcal{W}_j$ precedes T_1. This proves the existence of a chain of at least $p+1$ tasks of $\mathcal{W}_j$ and thus

$$(6) \qquad p + 1 \leq \omega_{\text{opt}}(\mathcal{W}_j).$$

On the other hand, if each task T of $\mathcal{W}_j$ executed before T_1 is such that $\alpha(T) \geq \alpha(T_1)$, then there exists at least $m+1$ tasks of $\mathcal{W}_j$ preceding a chain of at least $p-1$ tasks of $\mathcal{W}_j$ and the inequality (6) still holds.

Now we consider that the first slot of $\lambda(\mathcal{W}_j)$ is partial and establish a lower bound of $\text{Idle}_{\text{opt}}(\mathcal{W}_j)$. The existence of a chain including p tasks of $\mathcal{W}_j$ leads to

$$(6') \qquad p \leq \omega_{\text{opt}}(\mathcal{W}_j).$$

Let us assume that ${}^jU$ is not executed during λ_1. Then, each task of $\mathcal{Y}_j \backslash \mathcal{F}$ is preceded by at least one task of $\mathcal{F}$ for the ECG-algorithm computes a list schedule. Furthermore, every task of $\mathcal{A}$ is preceded by a task of $\mathcal{Y}_j$ and it is easy to verify that ${}^jU$ precedes all the tasks of $\mathcal{X}\backslash\mathcal{Y}_j$. Hence, each task of $\mathcal{W}_j\backslash\mathcal{F}$ is preceded by a task of $\mathcal{F}$ and thus

$$\text{(7)} \qquad \text{Idle}(\mathcal{F}) \leq \text{Idle}_{\text{opt}}(\mathcal{W}_j).$$

If ${}^jU$ is executed during λ_1 then $\mathcal{W}_j$ is equal to $({}^jU)$ and the inequality (7) still holds.

- The inequality (8) is derived from the usual expressions $\omega(\mathcal{W}_j) = (|\mathcal{W}_j|+\text{Idle}(\mathcal{W}_j))/m$ and $(|\mathcal{W}_j| + \text{Idle}_{\text{opt}}(\mathcal{W}_j))/m = \omega_{\text{opt}}(\mathcal{W}_j)$ and from the inequality (5) and (6), or from the inequality (5′), (6′), and (7) depending on whether the first slot of $\lambda(\mathcal{W}_j)$ is partial or not:

$$\text{(8)} \qquad \omega(\mathcal{W}_j) \leq (2-2/m)\omega_{\text{opt}}(\mathcal{W}_j) - (m-3)/m.$$

The inequality (8) is refined by factorizing its righthand member, which becomes equal to

$$\text{(8)} \qquad 2\omega_{\text{opt}}(\mathcal{W}_j) - 1 - \frac{2\omega_{\text{opt}}(\mathcal{W}) - 3}{m}.$$

When m is even, the term $(2\omega_{\text{opt}}(\mathcal{W}) - 3)/m$ cannot be integer, and hence (8) can be changed into (8′):

$$\text{(8}'\text{)} \qquad \omega(\mathcal{W}_j) \leq (2-2/m)\omega_{\text{opt}}(\mathcal{W}_j) - (m-2)/m.$$

Figure 5 presents the worst case leading to (8) or (8′). The local-bound lemma ensues from these facts (*odd* (m) is equal to 1 if m is odd and is equal to 0 otherwise).

LOCAL-BOUND LEMMA. *Let a schedule of* $(\mathcal{C}, \ll)$ *be computed on m machines by the* ECG-*algorithm and* $\mathcal{W}$ *be a segment with respect to this schedule. If* $m \geq 2$, *then*

$$\omega(\mathcal{W}) \leq \left(2 - \frac{2}{m}\right) w_{\text{opt}}(\mathcal{W}) - \frac{m - 2 - \text{odd}(m)}{m}$$

and the equality is achieved in the worst case.

6.5. The worst-case bound. We remain in the context of the subsection 6.4. If the critical block $\mathcal{Y}_{j+1}$ exists, the segment $\mathcal{W}_j$ precedes $\mathcal{Y}_{j+1}$ by definition. Furthermore, the critical task of $\mathcal{Y}_{j+1}$ precedes each task of the blocks to its right and each additional task of $\mathcal{W}_{j+1}$ is preceded by a task of $\mathcal{Y}_{j+1}$. Thus $\mathcal{W}_j \ll \mathcal{W}_{j+1}$ and the segments make up a chain. The local-bound lemma can hence be extended to the entire set of tasks:

THEOREM. *Let a schedule of* $(\mathcal{C}, \ll)$ *be computed on m machines by the* ECG-*algorithm. If* $m \geq 2$, *then*

$$\omega(\mathcal{C}) \leq \left(2 - \frac{2}{m}\right) \omega_{\text{opt}}(\mathcal{C}) - \frac{m - 2 - \text{odd}(m)}{m}$$

and the equality is possible.

The analysis we have presented still holds if the set $\mathcal{S}(T)$ includes all the successors of T rather than its immediate successors only. The only point that is different if $\mathcal{S}(T)$ includes all the successors of T is the proof of the segment properties that is based on the segment definition. It becomes simpler. Therefore, if the transitive closure of the graph to be scheduled is available, it is not necessary to compute the transitive reduction of the graph.

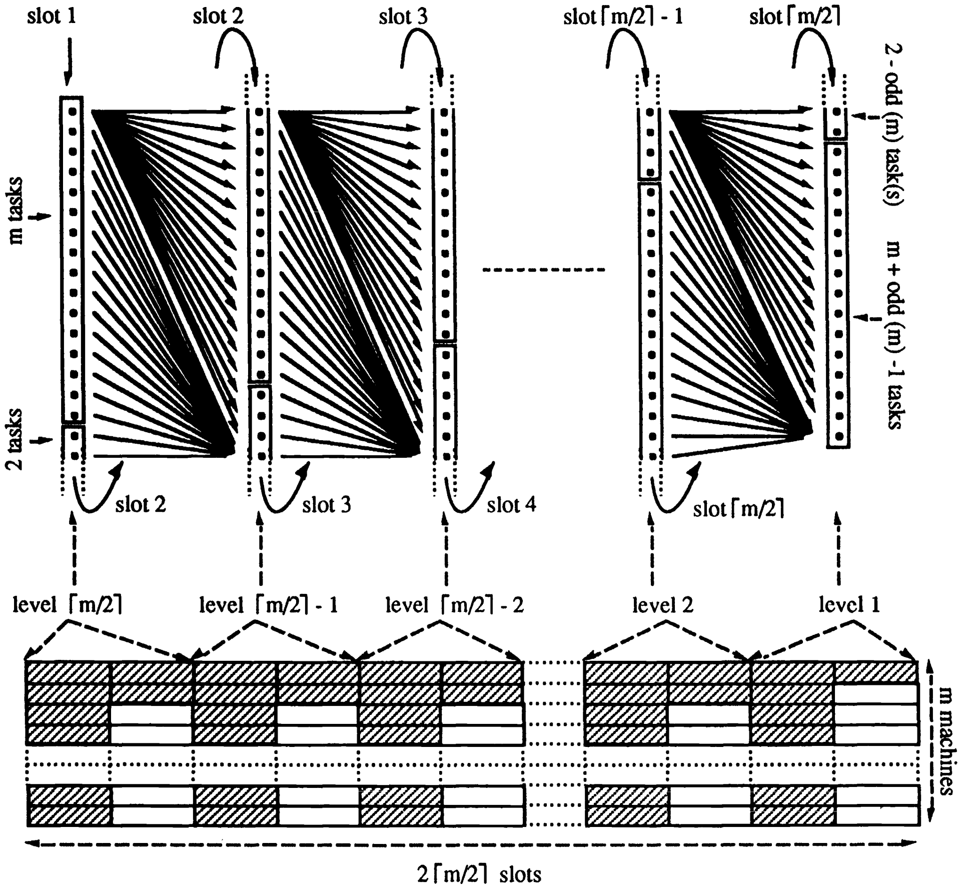


FIG. 5. *Worst-case instance of the local-bound lemma. The graph is made up of* $\lceil m/2 \rceil$ *levels, following a basic critical-path analysis. The first level is composed of* $m + 1$ *tasks and all other levels include* $m + 2$ *tasks. The task at the top of the figure on level* $i, i > 1$ *precedes all the tasks on level* $i - 1$ *except the task at the bottom. On the one hand, the optimal schedule on m machines is sketched directly on the graph: the boxes show the tasks executed during a particular slot. The length of the optimal schedule is* $\lceil m/2 \rceil + 1$*. On the other hand, we give priorities to the task according to the CG-rule and we break the ties between two tasks by giving the lowest priority to the topmost task. In each level, the priorities are decreasing from the bottom to the top. The induced schedule length is* $2\lceil m/2 \rceil$.

REFERENCES

[1] M. BARTUSCH, R. H. MOHRING AND F. J. RADERMACHER, *m-machine unit time scheduling: a report on ongoing research*, in Optimization and Parallel Processing, Kurzenski, Neumann, and Pallaschke, eds., Springer-Verlag, New York, 1987, pp. 165–212.

[2] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta Inform., 1 (1972), pp. 200–213.

[3] M. FUJII, T. KASAMI, AND K. NINOMIYA, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[4] H. N. GABOW, *An almost linear algorithm for two-processor scheduling*, J. Assoc. Comput. Mach., 29 (1982), pp. 766–780.

[5] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[6] S. LAM AND R. SETHI, *Worst-case analysis of two scheduling algorithms*, SIAM J. Comput., 6 (1977), pp. 518–536.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 671–700, August 1994

A NEW APPROACH TO STABLE MATCHING PROBLEMS*

ASHOK SUBRAMANIAN†

Abstract. It is shown that Stable Matching problems are the same as problems about stable configurations of X-networks. Consequences include easy proofs of old theorems, a new simple algorithm for finding a stable matching, an understanding of the difference between Stable Marriage and Stable Roommates, NP-completeness of Three-party Stable Marriage, CC-completeness of several Stable Matching problems, and a fast parallel reduction from the Stable Marriage problem to the Assignment problem.

Key words. stable matching, stable marriage, stable roommates, stable configuration, circuit value, network stability, parallel complexity, comparator, CC, scatter, assignment problem, weighted matching

AMS subject classifications. 05A05, 68Q10, 68Q15, 68Q20, 68R05, 90C27, 94C99

1. Introduction.

In a Stable Matching problem, the task is to match a number of persons in pairs, subject to certain preference information. Briefly, each person regards some of the others as acceptable mates and ranks them in order of preference. A matching is unstable if there are two persons, not matched to each other, who would rather be together. The task is to find a stable matching, i.e., one that is not unstable.

This problem was first studied by Gale and Shapley [7]. They showed that a stable matching always exists if the problem is a Marriage problem, i.e., if the participants can be divided into two sexes, the men and the women, in such a way that the acceptable mates of each person are all of the opposite sex; in fact, they gave a linear-time algorithm to find such a matching. Irving, in [14], gave a linear-time algorithm for the general problem. An introductory treatment of Stable Matching appears in [23]; a comprehensive treatment may be found in [12].

This paper explores the relationship between Stable Matching and Network Stability. For our purposes, a network is a boolean circuit with feedback. The task in Network Stability is, given a network and its inputs, to find an assignment of boolean values to the edges of the network that respects the input constraints and the gate equations. The Network Stability problem is NP-complete in general [21], but when every gate in the network is a special gate called the X-gate, the problem becomes equivalent to Stable Matching. The X-gate has a certain nice property—it is *scatter-free*. We exploit this property to solve X-Network Stability efficiently.

The appropriateness of the new framework is illustrated by new easy proofs of old theorems and simple algorithms. We give simple linear-time algorithms for finding a stable matching and for finding a "minimum-regret" stable matching. For both of these problems, optimal algorithms were already known [14], [15]. We give an easy proof of the theorem of Gale and Sotomayor [8] that every stable matching of a given instance matches the same set of persons. We also prove a structural theorem that might help us understand why certain instances of Stable Matching have no solutions. We give a new simple proof of the #P-completeness of counting the number of solutions to an instance of Stable Matching; this result, in a stronger form, was first proven in [16]. It turns out that just as Stable Matching is equivalent to Network Stability for X-networks, a certain "three-party" version of Stable Matching is equivalent to Network Stability for Y-networks. We use properties of the Y-gate to

*Received by the editors August 9, 1989; accepted for publication (in revised form) October 20, 1992. This work was supported by Office of Naval Research contract N00014-88-K-0166.

†Department of Computer Science, Stanford University, Stanford, California 94305-2140. Current address: Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

conclude that Three-party Stable Marriage is NP-complete, thus partly answering a question posed by Knuth [20].

It has been observed repeatedly that the Stable Marriage problem seems easier and has more structure than Stable Matching. We explain this difference by observing that the networks corresponding to Stable Marriage are comparator networks. The comparator gate is a simpler gate than the X-gate. The monotonicity of the comparator is responsible for the fact that Stable Marriage instances always have solutions. Also, Conway's lattice theorem for Stable Marriage can be viewed as a direct consequence of the lattice theorem for comparator networks.

We also study the parallel complexity of Stable Matching. While we do not give any fast parallel algorithms for the problem, we do point out that several problems related to Stable Matching are complete for the class CC of problems defined in [21]. We use these ideas to give a fast parallel reduction from the Stable Marriage problem to the Assignment problem, thus partially solving a problem of Knuth [20].

Our approach does not seem to apply when the instances of Stable Matching have ties or when issues of deceit are involved. It does not explore the structure of all stable matchings of an instance, as does the work of Gusfield [10], [11], Irving [15], [17], or Feder [5].

2. Preliminaries.

2.1. Stable Matching.

An instance of Stable Matching (also called Stable Roommates) consists of a set of persons, each of whom regards some of the others as acceptable mates, and ranks them in decreasing order of preference. For our purposes we may assume that acceptability is mutual—x is acceptable to y if and only if y is acceptable to x. A matching is a pairing of some or all of the persons. A matching may be *unstable* in three ways—two unmatched persons may find each other acceptable; a matched person may prefer an unmatched person to his current mate; or two matched persons may prefer each other to their current mates. A matching that is not unstable is said to be *stable*.

An instance of Stable Matching is an instance of Stable Marriage if the persons can be divided into two sets, the men and the women, so that the acceptable mates of each person are all of the opposite sex. An instance of Stable Matching is an instance of Complete Stable Matching if there are an even number of persons and each person is acceptable to everyone else. Similarly, an instance of Stable Marriage is an instance of Complete Stable Marriage if there are an equal number of men and women and each person is acceptable to every person of the opposite sex.

The size of an instance of Stable Matching is the sum, over all persons x, of the number of persons acceptable to x. The most common tasks associated with an instance of Stable Matching are to determine whether a stable matching exists and to construct one if possible. Other tasks might include counting and enumerating all stable matchings of a given instance.

Remark. Historically, most of the interest in the literature has been in Complete Stable Marriage and Complete Stable Matching; these problems are commonly called the Stable Marriage and Stable Roommates problems, respectively. Also, it has been traditional to measure the running time of algorithms in terms of the number of participants and not in terms of the "length of the input." Our departure from this tradition might lead to one cause of confusion: algorithms that we describe as linear time (in terms of the size of the instance) might be described in the literature as taking quadratic time (in terms of the number of participants).

2.2. Gates, circuits, and networks.

The definitions in this section and in §2.3 are taken almost entirely from [21].

A λ-input, μ-output *gate* is a function $g : \{0, 1\}^\lambda \rightarrow \{0, 1\}^\mu$ from λ-bit input words to μ-bit output words. In this paper, all gates have a fixed number of inputs and outputs. Further, a gate does not have useless inputs or outputs—each input bit affects some output bit, and each output bit depends nontrivially on some input bit. A *basis* is a set of gates. The gates that appear in this paper are listed in Table 1.

TABLE 1
The gates that appear in this paper.

Gate	Inputs	Outputs
ID	i_1	$o_1 = i_1$
COPY	i_1	$o_1 = i_1,\ o_2 = i_1$
NOT	i_1	$o_1 = \overline{i_1}$
AND	i_1, i_2	$o_1 = i_1\, i_2$
OR	i_1, i_2	$o_1 = i_1 + i_2$
NAND	i_1, i_2	$o_1 = \overline{i_1} + \overline{i_2}$
C	i_1, i_2	$o_1 = i_1\, i_2,\ o_2 = i_1 + i_2$
X	i_1, i_2	$o_1 = i_1\, \overline{i_2},\ o_2 = \overline{i_1}\, i_2$
Y	i_1, i_2, i_3	$o_1 = i_1\, (\overline{i_2} + \overline{i_3}),\ o_2 = i_2\, (\overline{i_1} + \overline{i_3}),\ o_3 = i_3\, (\overline{i_1} + \overline{i_2})$

A *network* is a finite labeled directed graph. Source (in-degree zero) nodes of the directed graph have out-degree one and are called input nodes; sink (out-degree zero) nodes have in-degree one and are called output nodes. Each internal node is labeled with a gate and an ordering of its predecessors and successors. If an internal node has in-degree λ and out-degree μ, its gate has λ inputs and μ outputs. If the underlying directed graph of a network is acyclic, the network is called a *circuit*. A circuit computes a set of functions in the standard manner. A network (circuit) is said to be *over* a basis Ω if every gate in it is from Ω. The *size* of a network is the number of edges in it.

Our definition of circuit differs from the standard definition in two respects—all fanout is explicit and must occur within a gate, and multiple-output gates are allowed. Our definition of "network" is nonstandard. A network is essentially a combinational circuit with feedback; it is not intended to compute anything.

A gate g_1 is a *restriction* of gate g_2 if it can be obtained as follows. Fix a subset (possibly empty) of the inputs of g_2 to constant values. This operation fixes a (possibly empty) set of outputs of g_2 to constant values. Now discard all the inputs and outputs of g_2 that are fixed and perhaps some outputs that are not fixed. For example, the *NOT*-gate is a restriction of the X-gate, obtained by fixing any one input of the X-gate to 1. Also, the *closure* of any basis Ω is the basis Ω^* consisting of all restrictions of gates in Ω.

Gate g can be *simulated* by basis Ω if g is a restriction of the function computed by some circuit over Ω. We say, for convenience, that gate g_1 can be simulated by gate g_2 when we mean g_1 can be simulated by the basis $\{g_2\}$; similarly, we say a basis can be simulated (by another basis) if every gate in it can be simulated. For instance, a gate (or basis) is monotone if it cannot simulate $\{$*NOT*$\}$.

A gate has *scatter* if it has a restriction that has more outputs than inputs; otherwise, it is *scatter-free*. A basis is said to be scatter-free if every gate in it is scatter-free. All the gates in Table 1, except the *COPY*-gate and the Y-gate, are scatter-free.

The X-gate and the C-gate (comparator) play crucial roles in this paper. These gates are scatter-free; in addition, the comparator is monotone. Many of the results of this paper rely on these elementary facts.

The following easy lemma is crucial.

LEMMA 2.1. *Let N be any network over a scatter-free basis Ω. Then N has at most as many outputs as inputs. In particular, if N has no inputs, it has no outputs.*

Proof. Sum the quantity (#(inputs of g) $-$ #(outputs of g)) over all gates g of N. Each input edge of N is an input edge of exactly one gate of N and thus contributes one to the sum;

similarly, each output edge of N contributes minus one. Each internal edge of N is an input edge of exactly one gate of N and an output edge of exactly one gate of N and so contributes zero. Thus the sum equals the difference between the number of inputs of N and the number of outputs of N. Notice, however, that the sum is nonnegative, because each (scatter-free) gate of N contributes a nonnegative quantity to the sum; hence N has at most as many outputs as inputs. □

2.2.1. X-networks and snakes in X-networks. An X-network is a network over $\{X\}$. We introduce some terminology to describe X-networks. Recall that the X-gate has inputs i_1, i_2 and outputs $o_1 = i_1 \overline{i_2}$, $o_2 = \overline{i_1} i_2$. We *associate* input i_j and output o_j. This association allows us to describe an X-network in terms of *snakes*. A snake is a sequence of edges; the sequence may be cyclic or acyclic. If an input edge of an X-gate is in snake q, the associated output edge is the next element of q; if an output edge of an X-gate is in q, the associated input edge is the previous element of q. The first element of an acyclic snake is thus an input edge of the network; the last element of an acyclic snake is an output edge of the network. The snakes of a network constitute a partition of its edges.

We often describe an X-network by naming its snakes and telling how they meet each other; in such a description, the gates are implicit—there is a gate wherever two snakes meet. We say that a snake enters a gate with value v if the edge of the snake that is an input to the gate has the value v. (Some care is needed, because a snake could enter the same gate twice, once through each input.)

2.3. The Circuit Value and Network Stability problems. The Circuit Value problem (CV) is the task of computing the value assigned to a specified output of a circuit, when the circuit is evaluated on a given input assignment. More formally, given a circuit with λ inputs and an input assignment $s_{\text{in}} \in \{0, 1\}^\lambda$ to the circuit, determine whether the specified output takes the value 1.

The Network Stability problem is a question about the existence of *configurations* of a network consistent with a given input assignment. Let N be a network with $\lambda \geq 0$ inputs. Let $s_{\text{in}} \in \{0, 1\}^\lambda$ be an input assignment to N. A *configuration* Q of N is an assignment of boolean values (0s and 1s) to the edges of N. It is a *stable configuration* of $[N, s_{\text{in}}]$ if it satisfies the gate equations at each internal node and is consistent with the input assignment s_{in}. The *Network Stability problem* (NS) is this: given a network N and an input assignment s_{in}, determine whether $[N, s_{\text{in}}]$ has a stable configuration. Often we leave the input assignment implicit, and say "N has a stable configuration" when we mean "$[N, s_{\text{in}}]$ has a stable configuration."

Given a basis Ω, we define Ω-CV and Ω-NS to be the special cases of CV and NS, respectively, in which the network is required to be over Ω. For instance, C-CV is the circuit value problem for comparator circuits, and X-NS is the stability problem for X-networks.

The input assignment that assigns every input of a network the value 1 is called **1**; the empty input assignment is called **e**.

3. The Stable Matching problem is an X-Network Stability problem. We show how to transform an instance I of Stable Matching into an X-network N in such a way that the stable matchings of I are in one-to-one correspondence with the stable configurations of $[N, \mathbf{1}]$. Here is the transformation. Network N has one acyclic snake q_x for each person x of instance I; there are no other snakes. Snakes q_x and q_y meet if and only if x and y find each other acceptable; snake q_x meets the snakes of other persons in the order that these persons appear on the preference list of person x, i.e., in decreasing order of their popularity with x. Any two snakes meet at most once; no snake meets itself. This description completely specifies N. We present a method to construct N given the description. For each pair x, y of persons who find each other acceptable, introduce an X-gate labeled with the pair $\{x, y\}$. For each person, introduce a snake that visits all the gates whose labels contain his name, in the order determined by his preference list. This completes the construction. An example is shown in Fig. 1. (In this figure, the snake of each person is labeled with his name for clarity.)

Person	Preferences		
A	D	B	
B	D	C	A
C	B	D	
D	A	B	C

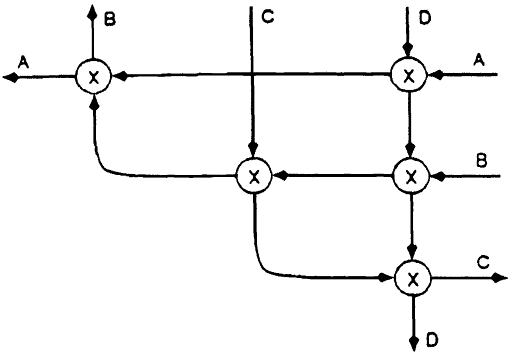


FIG. 1. *Reducing Stable Matching to X-Network Stability.*

In §3.1, we prove a lemma about stable configurations of X-networks. We then prove the correspondence between stable matchings and stable configurations in §3.2.

3.1. A preliminary lemma about stable configurations of X-networks. Let N be any X-network without cyclic snakes. Let Q be a stable configuration of $[N, \mathbf{1}]$. The stability of Q places two conditions upon it—it must satisfy the input assignment, and it must satisfy all the gate equations. Let us investigate what these conditions really mean. Let $q = \langle e_1, e_2, \ldots \rangle$ be any snake of N; for each j, let v_j be the value on edge e_j. Every input edge has the value 1, so $v_1 = 1$. The behaviour of the X-gate requires that $v_j = 0 \Rightarrow v_{j+1} = 0$. Thus, the sequence $\langle v_1, v_2, \ldots \rangle$ is a sequence of 1s, followed perhaps by a sequence of 0s. Define the *1-segment* of snake q in configuration Q to be the subsequence of edges that have value 1.

We say that snake q *drops* at gate g if the 1-segment of q ends at gate g; we say that g is the endpoint of the 1-segment of q. (A snake that has the value 1 along its entire length is identical with its 1-segment and has no endpoint.) It is clear that each snake drops at most once.

A snake that enters a gate with value 1 will drop if and only if the other snake entering the gate also enters with value 1. Thus, snakes must drop in pairs. Also, the 1-segments of different snakes cannot share vertices, except possibly a common endpoint. We describe this situation informally by saying that 1-segments avoid one another.

It turns out that the conditions listed above are necessary and sufficient for Q to be a stable configuration. In other words,

LEMMA 3.1. *Let N be an X-network without cyclic snakes; let Q be a configuration of* $[N, \mathbf{1}]$. *Then Q is stable if and only if:*

(i) *the values on any given snake of N form a nonempty sequence of* 1*s, followed by a possibly empty sequence of* 0*s,*

(ii) *snakes drop in pairs, and*

(iii) *the* 1*-segments of different snakes avoid one another.*

Proof. We have already seen that the above three conditions will be met if Q is stable. Suppose now that Q is a configuration that meets the three conditions; we show that Q must be stable. The first value on each snake is a 1, so the input conditions are satisfied. Let g be an arbitrary gate of N; we must show that the gate equations are satisfied at g. If both snakes entering g do so with value 0, condition (i) requires that both outputs of g be 0 as well. If one snake is 0 and the other is 1 while entering g, conditions (i) and (ii) demand that both snakes leave g with the same values with which they entered it. If both snakes are 1 while entering g, condition (iii) demands that both drop. In all three cases, the gate equations are satisfied. □

3.2. The correspondence between stable matchings and stable configurations.

THEOREM 3.2. *Let I be an instance of Stable Matching; let N be the corresponding X-network. Then there is a one-to-one correspondence between the stable matchings of I and the stable configurations of* $[N, \mathbf{1}]$.

Proof. The rule to read off a matching of I from a stable configuration of $[N, \mathbf{1}]$ is this: two persons are matched to each other if their snakes drop together; a person is unmatched if his snake does not drop. To produce the configuration of $[N, \mathbf{1}]$ corresponding to a given matching of I, we invert the above rule: if person x is matched to person y, snake q_x takes the value 1 until it meets snake q_y and the value 0 thereafter; if x is unmatched, snake q_x is 1 along its entire length. We now show that stable configurations yield stable matchings and vice-versa.

Suppose that stable configuration Q yields matching M. Assume that M is unstable. Hence there must be two persons x and y, each of whom prefers the other to his mate in M or to the state of being unmatched. Since x and y are acceptable to each other, snakes q_x and q_y meet. We claim that q_x is 1 after meeting q_y. To prove the claim, we consider two cases. If x is unmatched, then q_x must be 1 along its entire length, and we are done. In the other case, let z be the mate of x. Then q_x and q_z must meet with value 1; hence q_x must be 1 all the way until it meets q_z. Since x prefers y to z (by hypothesis), q_x meets q_y *before* it meets q_z. It follows that q_x is 1 after meeting q_y; this proves the claim. By an entirely analogous argument, q_y is 1 after meeting q_x. This means that the 1-segments of q_x and q_y do not avoid one another. Lemma 3.1 tells us that Q must be unstable, which is a contradiction. Hence M must be stable.

Now suppose that stable matching M yields configuration Q. It is clear that Q meets conditions (i) and (ii) of Lemma 3.1. To prove that Q is stable, it suffices to show that condition (iii) is met as well. Assume, for the sake of contradiction, that there are two persons whose snakes violate condition (iii). Then it follows that each of these persons prefers the other to his mate (if any) in M, contradicting the stability of M. □

Remark. We reiterate the meaning of the value assigned to an edge by a stable configuration of $[N, \mathbf{1}]$. Let e be the edge of snake q_x just after q_x meets q_y. Then e is assigned the value 0 in a stable configuration of $[N, \mathbf{1}]$ if and only if x has a mate no worse than y in the corresponding stable matching of I.

4. Solving Network Stability for scatter-free networks.

4.1. The algorithm. In this section, we show how to construct a stable configuration of a network over any scatter-free basis. This algorithm was presented in [21]. The X-gate is scatter-free, so the results of this section apply to Stable Matching. In particular, we get a simple linear-time algorithm for constructing a Stable Matching. A different linear-time algorithm for this problem was already known [14].

Let N be any network over a scatter-free basis Ω; let s_{in} be an input assignment to N. We show how to find a stable configuration of $[N, s_{\text{in}}]$. The interesting case is when N has no inputs, i.e., when s_{in} is the null assignment $\mathbf{e}$. Otherwise, we could use the input assignment to simplify the network. After all, if we know the value of an input to a gate g, we can replace g by a restriction with fewer inputs and perhaps deduce the value of some of the outputs of g. Iterating this process of "input elimination" yields a partial assignment P

of values to edges and an input-free network N' on the remaining edges, with the property that the stable configurations of $[N, s_{\text{in}}]$ are obtained by augmenting the stable configurations of $[N', \mathbf{e}]$ with P. In particular, $[N, s_{\text{in}}]$ has a stable configuration if and only if $[N', \mathbf{e}]$ does. Notice that the time taken to perform input elimination is linear in the amount of simplification performed, i.e., in the number of edges that are assigned values by the partial assignment P. The order in which the inputs are eliminated is immaterial; in fact, the process is properly regarded as a concurrent process, although we do not use this fact here. Lemma 2.1 assures us that N' will have no outputs. Henceforth we will simply assume that N has no inputs or outputs.

Consider what happens if we break an edge e of N, thus creating an input $e(\text{in})$ and an output $e(\text{out})$, and then place the value $v \in \{0, 1\}$ on $e(\text{in})$. We get a network with one input and one output; we may apply input elimination to this network. We call this the process of *propagating* the pair $[e, v]$. The following observation is crucial: the elimination of the sole input of the network must assign a value, say $v(\text{out})$, to the output edge $e(\text{out})$. To see this, apply Lemma 2.1 to the network that remains after the propagation. If $v(\text{out}) = v$, we say that the propagation is *successful* and that $[e, v]$ is *viable* in N. The result of a successful propagation is a partial assignment P and an input-free network N' on the remaining edges, with the property that the stable configurations of $[N, \mathbf{e}]$ that assign edge e the value v are obtained by augmenting the stable configurations of $[N', \mathbf{e}]$ with P. If, on the other hand, $v(\text{out}) \neq v$, no stable configuration of $[N, \mathbf{e}]$ may assign edge e the value v; in this case, we say the propagation is *unsuccessful*. Again, the time taken by a successful propagation is linearly related to the amount of simplification achieved.

The *type* of an edge in an input-free network is the set of viable values for it, i.e., $\text{type}_N(e) = \{v \mid [e, v] \text{ is viable in } N\}$. If $\text{type}_N(e) = \emptyset$, we say that e is a *contradicting* edge. Clearly, if the edge we select is contradicting, there can be no stable configurations. Also, if its type is $\{0\}$ or $\{1\}$, there is only one possible value that this edge could take in any stable configuration; thus we can assign this edge its value and simplify the network accordingly. The difficulty arises when both values seem possible, i.e., when $\text{type}_N(e) = \{0, 1\}$.

Our strategy in such a situation is to pick *either* value and proceed. Clearly, if we are lucky, we will make all the right choices and find a stable configuration. But what if we are unlucky? Might not a set of bad choices early on in the algorithm lead us to a situation where we can no longer complete the partial assignment we have to a stable configuration? Lemma 4.1 comes to our rescue, because it says that the type of an edge is preserved by successful propagations. This means that if we find a contradicting edge after performing a sequence of successful propagations, then the same edge must have been contradicting in the original network as well, meaning that there were no stable configurations from the start. In other words, there are no bad choices.

LEMMA 4.1. *Let N be an input-free network over a scatter-free basis Ω. Suppose that $[e_1, v_1]$ is viable in N. Let N' be the network left after propagating $[e_1, v_1]$ in N; let e_2 be an edge of N'. Then* $\text{type}_{N'}(e_2) = \text{type}_N(e_2)$.

Proof. This proof is due to Feder [6]; our original proof was much more complicated. We make the following definition. A *trace* of the propagation of $[e_0, v_0]$ in N is a sequence of statements that records all the assignments that are made during this propagation, along with the dependencies between the assignments. Each statement is of the form $e \leftarrow v$, meaning "edge e gets value v." The first statement is $e_0 \leftarrow v_0$ and constitutes the *preamble* of the trace; the rest of the trace is called its *body*. A statement in the body of the trace is said to be *terminal* if it mentions edge e_0. All other statements of the trace are *nonterminal*. Every statement in the body is derived from a subset of the nonterminal statements preceding it in the trace by invoking the equation of a single gate. The trace is complete, in the sense that all statements that are derivable in the above manner are included. A statement may occur at most once in the body of the trace; any subsequent occurrences would be redundant.

Each statement of a trace may be regarded as a step in the propagation process. A propagation succeeds if the statement $e_0 \leftarrow v_0$ appears in the body of the trace; it fails if the statement $e_0 \leftarrow \overline{v_0}$ appears in the body. Clearly both statements cannot appear simultaneously in the body of any trace; on the other hand, we know that at least one of these statements will appear in the body, because the network is input-free and scatter-free.

Let T_1 be a trace of $[e_1, v_1]$ in N; let T_2 be a trace of $[e_2, v_2]$ in N. The effect of propagating $[e_1, v_1]$ is to assign values to some edges of N. We would like to claim that, even after these assignments are made, each statement in the body of T_2 can still be inferred from the preamble of T_2. This would mean that propagating $[e_2, v_2]$ in N' will establish every assignment that would have been made by the propagation of $[e_2, v_2]$ in N that has not already been made by the propagation of $[e_1, v_1]$ in N. It would then follow that $[e_2, v_2]$ is viable in N if and only if it is viable in N', completing the proof of Lemma 4.1.

To prove the claim, it suffices to show that there is no conflict between the two traces. A conflict is a situation where the statement $e \leftarrow v$ appears in one trace while the statement $e \leftarrow \overline{v}$ appears in the other. We assume a conflict exists and derive a contradiction. Let S_2 be the earliest statement in T_2 that is involved in a conflict; assume that it says $e_c \leftarrow v_c$ and that it conflicts with statement S_1 of T_1. Statement S_2 cannot be in the preamble of T_2, because, by hypothesis, edge e_2 does not appear in T_1. Hence, edge e_c is the output of some gate; let this gate be g_c. The gate equations used to derive S_1 and S_2 assert that g_c has two partial input assignments P_1 and P_2, such that on one of these assignments, output e_c takes the value v_c, whereas on the other assignment, output e_c takes the value $\overline{v_c}$. Now, by the choice of S_2, the partial input assignments P_1 and P_2 do not conflict. Let P be the partial input assignment that is the union of P_1 and P_2. Consider the behaviour of gate g_c on P. Clearly, output e_c must be both 0 and 1 in this situation, which is a contradiction. □

An immediate consequence of Lemma 4.1 is the correctness of the following simple polynomial-time algorithm for constructing a stable configuration of a network over any scatter-free basis.

ALGORITHM 1.

1. Eliminate all inputs.
2. Pick any edge e. (If there are no edges, there is nothing to do.)
3. Determine that e is contradicting (in which case there are no stable configurations), or find a value v such that $[e, v]$ is viable.
4. Suppose $[e, v]$ is viable. Propagate $[e, v]$, obtaining a partial assignment P and a simplified network N'.
5. Find a stable configuration of N' and augment it with P to yield the desired stable configuration. If N' has no stable configuration, neither does N.

THEOREM 4.2. *Let N be a network over a scatter-free basis Ω; let s_{in} be any input assignment to N. There is a linear-time algorithm that will construct a stable configuration of $[N, s_{\text{in}}]$ if there is one, or conclude that none exists.*

Proof. Algorithm 1 can be implemented to run in time that is linear in the size (number of edges) of the network N. Notice that steps 1 and 4 take time linear in the amount of simplification they perform and that step 2 takes constant time. Also, if step 3 determines that e is contradicting, the time needed to do so is linear in the size of the network. The only obstacle to a linear run-time is that step 3 may take a long time to return a viable value v. To avoid this, we use the following standard trick. Implement step 3 as follows: propagate $[e, 0]$ and $[e, 1]$ "in parallel"; whichever propagation succeeds first determines the value v to be used in step 4. This guarantees that step 3 runs in time linear in the amount of simplification performed in step 4. The linear time bound for the algorithm follows from the observation that the size of N' in step 5 is less than the size of N by the amount of simplification achieved in step 4. □

The following corollary is useful in the context of parallel computation:

THEOREM 4.3. *Let N be an input-free network over a scatter-free basis Ω. Then N has a stable configuration if and only if it has no contradicting edge.*

Proof. If N has a contradicting edge, this edge can take neither the value 0 nor the value 1 in any stable configuration. Hence there cannot be any stable configurations. If, on the other hand, there are no contradicting edges, Algorithm 1 will find a stable configuration. □

The type of an edge has a nice interpretation:

THEOREM 4.4. *Let N be an input-free network over a scatter-free basis Ω; suppose that N has no contradicting edges. Let e be an edge of N. Then the* type *of e in N equals the set of values assigned to e in the (various) stable configurations of N.*

Proof. Clearly, if $[e, v]$ is not viable in N, then no stable configuration of N may assign e the value v. If $[e, v]$ is viable, on the other hand, we can construct a stable configuration that assigns e the value v, by letting $[e, v]$ be the first edge-value pair picked by Algorithm 1. □

The following theorem is proven for Stable Marriage in [8]; a proof for the Stable Matching case can be found in [12]. We give an easy alternative proof.

THEOREM 4.5. *The set of unmatched persons in any instance of Stable Matching is the same in every stable matching.*

Proof. Let I be an instance of Stable Matching; let N be the corresponding X-network. The process of eliminating the inputs of N assigns a value to every output edge of N. Hence every stable configuration of $[N, \mathbf{1}]$ has the same output assignment. The theorem follows by observing that a person is unmatched in a stable matching of instance I if and only if the output edge of his snake takes the value 1 in the corresponding stable configuration of $[N, \mathbf{1}]$. □

It is an interesting exercise to interpret Irving's original algorithm for Stable Matching [14] in the network framework. Phase 1 of Irving's algorithm achieves essentially the same effect as input elimination on the appropriate X-network. Each phase 2 reduction of Irving's algorithm either finds a contradicting edge (and concludes that there are no stable configurations) or finds a viable edge-value pair, propagates it, and simplifies the network accordingly. The specific method used by Irving to find the viable edge-value pair (the seek-cycle procedure) is sophisticated and uses specific properties of X-gates that are not shared by all scatter-free gates. The amount of simplification achieved is proportional to the length of the all-or-nothing cycle found, whereas the amount of time spent in a phase 2 reduction is proportional to the length of the all-or-nothing cycle plus the length of the "tail"; hence Irving uses the device of remembering the tail in order to achieve a linear-time algorithm.

4.2. Application to Minimum-regret Stable Matching. Algorithm 1 has two kinds of flexibility: edge e may be chosen to be any edge of the network, and there is some flexibility in the choice of value v. We have already demonstrated how to use the second kind of flexibility to make the algorithm run in linear time. Here we explore a different possibility.

Let I be any instance of Stable Matching. The *regret* of a participant in a stable matching of I is the position of his mate on his preference list. Thus, a participant who gets his first choice of mate has regret 1. Clearly, each participant would like to minimize his regret. In the Minimum-regret Stable Matching problem, we seek a stable matching that minimizes the largest regret among all the participants. A linear-time algorithm is given in [10]. However, this algorithm relies on nontrivial properties of the structure of all stable matchings of an instance of Stable Marriage; moreover, it works only for instances of Stable Marriage. Irving [15] gives a linear-time algorithm that works for all instances of Stable Matching; see [12] for an exposition. We give a different linear-time solution based on the network formulation.

We assume that I has a stable matching and that no person is unmatched in the stable matchings of I. The second assumption is legal because, by Theorem 4.5, the unmatched persons are the same in every stable matching; hence we can find these unmatched persons by input elimination in linear time, delete them, and find a minimum-regret matching of the remaining participants.

Define regret(I) to be the value of the largest regret in a minimum-regret stable matching of I. The crux of the problem is the computation of regret(I). We first show how to proceed if regret(I) is known and then show how to compute regret(I).

Suppose regret$(I) = r$. Delete from each person's list all but the first r persons, and then make the acceptability relation symmetric: if person x does not appear on the list of person y, delete y from the list of person x. Let us call the resulting instance I_r. It is easily checked that the stable matchings of I_r are precisely the minimum-regret stable matchings of I. A stable matching of I_r may be found in linear time by using the strategy outlined in the proof of Theorem 4.2.

We show how a modified version of Algorithm 1 can compute regret(I) in linear time. Let N be the X-network for I. Number each edge of N by its position on its snake: input edges get the number 1; output edges get big numbers. The value of the largest regret in any stable matching is given by the largest number of an edge, among all the edges with value 1 in the corresponding stable configuration. This means that the way to get a small regret is to assign the value 0 to the edges with big numbers.

Run Algorithm 1 with the following modifications. Whenever we have to pick an edge-value pair to propagate, we choose an edge e that has the largest number and assign it the value 0. Suppose edge e is the edge numbered k on the snake of person x. Then assigning e the value 0 amounts to guaranteeing that person x gets one of his first $(k-1)$ choices. If the propagation of $[e, 0]$ fails, there is no stable matching, consistent with the choices already made, that assigns x one of his first $(k-1)$ choices; hence regret(I) must be k. (The reason it cannot be larger than k is because at the time we assign edge e the value 0, we have already succeeded in assigning every edge with number larger than k the value 0.) Also, if the propagation assigns the value 1 to another edge with number k, then again the value of the minimum regret is k. We continue assigning the value 0 to more and more edges with large numbers, until ultimately one of the two exit conditions applies. Thus this algorithm will compute regret(I). To see that it takes linear time, observe that the time taken by all the successful propagations taken together is linear in the number of edges of N; also, there is at most one unsuccessful propagation, which can take at most linear time.

5. The structure of the network of contradicting edges. Theorem 4.3 says that if an instance of X-network stability has no stable configurations, the associated input-free network contains at least one contradicting edge. In this section, we prove that the contradicting edges form disjoint cycles, each containing an odd number of *NOT*-gates. We believe this is a step toward understanding why certain instances of Stable Matching do not have stable matchings.

LEMMA 5.1. *Let N be any input-free network over X^*. Suppose that $[e_1, v_1]$ is viable in N and that edge e_2 is assigned the value v_2 during the propagation of $[e_1, v_1]$. Then $[e_2, v_2]$ is viable in N.*

Proof. As in the proof of Lemma 4.1, let T_1 be a trace of $[e_1, v_1]$ in N and T_2 be a trace of $[e_2, v_2]$ in N. If $e_1 = e_2$, then $v_1 = v_2$ and there is nothing to prove; so assume that $e_1 \neq e_2$. Notice that the preamble of T_2 occurs in T_1, and every statement of T_1 occurs as a nonterminal statement in T_1. Hence we may show inductively that each statement of T_2 occurs in T_1. It follows that the value assigned to edge e_2 by the propagation of $[e_2, v_2]$ is v_2. □

Lemma 5.1 says that successful propagations do not assign values to contradicting edges. By repeatedly performing successful propagations, we can assign values to all noncontradicting edges. What is left is an input-free network $N_\emptyset$ over X^* consisting of exactly the contradicting edges.

LEMMA 5.2. *The network $N_\emptyset$ of contradicting edges does not contain any X-gates.*

Proof. Assume the opposite, that $N_\emptyset$ contains an X-gate g, with input edges i_1, i_2 and output edges o_1, o_2. We derive a contradiction.

Let T_1 be a trace of $[o_1, 0]$ in $N_\emptyset$. Since o_1 is contradicting, the statement $o_1 \leftarrow 1$ must appear in the body of T_1. In order for this to happen, statements $i_1 \leftarrow 1$ and $i_2 \leftarrow 0$ must also appear in T_1; hence so must $o_2 \leftarrow 0$.

Now let T_2 be a trace of $[o_2, 0]$ in $N_\emptyset$. Edge o_2 must be assigned a value by the propagation of $[o_2, 0]$; hence at least one of the edges i_1 or i_2 must be assigned a value during this propagation. Let S be the first statement in the body of T_2 that assigns a value to any of the edges incident upon gate g. Clearly, statement S assigns a value to either i_1 or i_2. We claim that S appears in T_1. To prove the claim, observe that the preamble of T_2 occurs in T_1 and that the derivation of S in T_2 does not involve edge o_1 in any intermediate step. Hence we may prove inductively that every statement involved in the derivation of S in T_2 occurs as a nonterminal statement in T_1, and the claim follows. Thus S must be either $i_1 \leftarrow 1$ or $i_2 \leftarrow 0$. In either case, the statement $o_2 \leftarrow 0$ will appear in the body of T_2. This indicates that $[o_2, 0]$ is viable, which is a contradiction. □

THEOREM 5.3. *Each connected component of $N_\emptyset$ is a directed cycle over the basis $\{ID, NOT\}$ with an odd number of NOT-gates.*

Proof. Follows from the fact that $N_\emptyset$ is input-free and contains no X-gates. Each cycle must have an odd number of NOT-gates in order for the edges on it to be contradicting. □

Remark. The results of this section extend naturally to the case where N is over any scatter-free basis, but the proof of Theorem 5.3 is more involved. (See [27] for details.)

6. The case of Stable Marriage.

In this section, we examine the X-networks arising from instances of Stable Marriage. We show that these networks of X-gates may be replaced by comparator networks. This observation, together with the monotonicity of the comparator, yield easy algorithms for certain problems associated with Stable Marriage.

6.1. Bipartitionable X-networks and comparator networks.

A network of X-gates is *bipartitionable* if its snakes can be two-colored so that snakes of the same color never meet; such a coloring is called a bipartition. In any bipartition, the color of an edge is defined to be the color of the snake to which it belongs. In this paper, whenever we say that a network is bipartitionable, we shall implicitly mean that a bipartition is given. The X-networks corresponding to instances of Stable Marriage are bipartitionable: color the snakes of the men red and the snakes of the women blue. This shows that instances of Stable Marriage may be solved by solving X-Network Stability for bipartitionable X-networks. The converse is also true; see §7.3.

We adopt the following conventions: in all diagrams containing X-gates, an input and its associated output appear opposite each other; in diagrams of bipartitionable X-networks, the red edges are drawn vertically and the blue edges are drawn horizontally; in all bipartitionable networks derived from instances of Stable Marriage, the red snakes belong to men and the blue snakes belong to women.

Given a bipartitionable X-network N and an input assignment s_{in}, we show how to transform them into a comparator network N' and an input assignment s'_{in} to N' in such a way that the stable configurations of $[N, s_{\text{in}}]$ and $[N', s'_{\text{in}}]$ are isomorphic. Bipartition the snakes of N into red and blue snakes. Replace every blue edge of the network by two *NOT*-gates in series. Regard every X-gate g to be part of a triplet of gates: g, the *NOT*-gate on the blue input of g, and the *NOT*-gate on the blue output of g. Each triplet is functionally equivalent to, and hence may be replaced by, a comparator; see Fig. 2(a). After these replacements, the only noncomparator gates in the network are the *NOT*-gates on the blue inputs and outputs. These *NOT*-gates just mean that the input and output assignments of the two networks do not match exactly: the blue bits have to be complemented. Delete the *NOT*-gates to get N'. To get s'_{in}, complement the blue input bits of s_{in}.

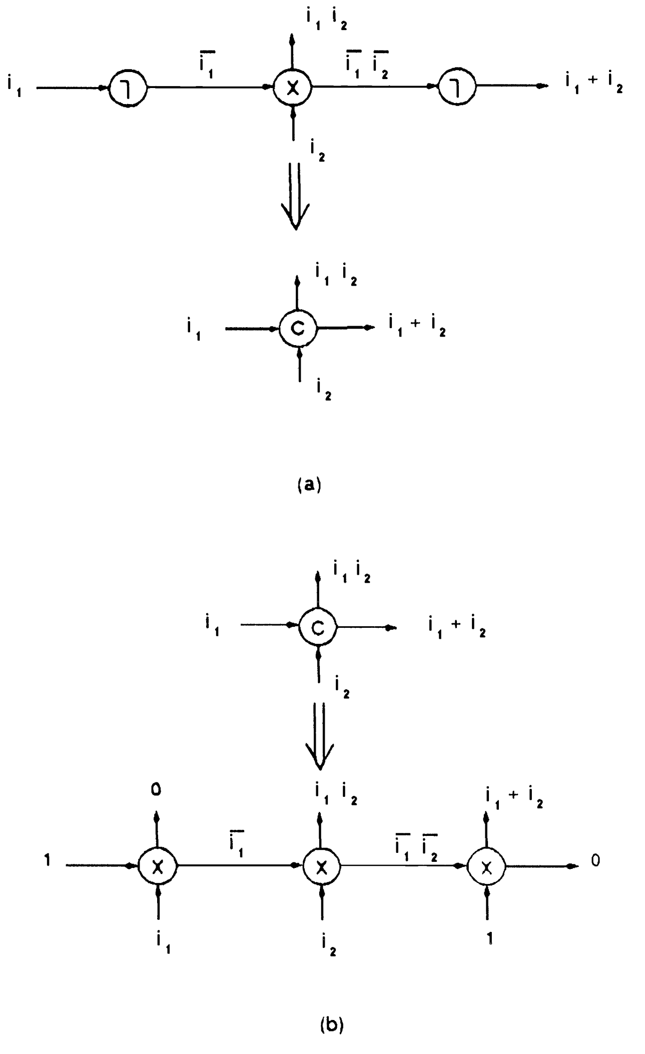


FIG. 2. *Interchanging comparators and X-gates.*

Essentially the same transformation may be used to replace a comparator network by a bipartitionable X-network on a different input assignment. Proceed as follows. Replace each comparator by three X-gates as shown in Fig. 2(b). A bipartition of the resulting X-network is obtained by coloring all the "horizontal" snakes blue and all the "vertical" snakes red.

These transformations show that finding stable configurations in bipartitionable X-networks is just the same as finding stable configurations in comparator networks. Thus, we may find stable matchings for instances of Stable Marriage by finding stable configurations of comparator networks.

6.2. Applications. In this section, we give linear-time algorithms for some problems related to Stable Marriage. For convenience of presentation, we phrase the following discussion partly in terms of bipartitionable X-networks and partly in terms of comparators.

6.2.1. Existence of stable matchings. The standard way to show that every instance of Stable Marriage has a stable matching is to give an algorithm, the proposal algorithm of [7], that finds the *man-optimal* stable matching in linear time. It is nice to be able to give a proof of the existence of stable matchings that does not involve an algorithm. Here is such a proof. An instance of Stable Marriage has a stable matching if and only if the associated comparator network has a stable configuration (on the appropriate input assignment). Comparator gates are monotone; monotone networks always have stable configurations.

6.2.2. Finding a stable matching. Finding a stable configuration of a monotone network is easy: eliminate the inputs, and then assign all remaining edges the value 0. This stable configuration is the "most-0" stable configuration Q^0 of the network (on the given input assignment), because every edge that is assigned the value 1 in this stable configuration must have the value 1 in every stable configuration. We show that this strategy yields the man-optimal stable matching found by the proposal algorithm of [7].

THEOREM 6.1. *Let I be an instance of Stable Marriage; let $[N, s_{\text{in}}]$ be the associated instance of Comparator Network Stability. Then the "most-0" stable configuration Q^0 of $[N, s_{\text{in}}]$ yields the man-optimal stable matching of I.*

Proof. Let $\tilde{N}$ be the X-network corresponding to I. (Thus, N is obtained from $\tilde{N}$ by applying the first transformation of §6.1.) The configuration of $[\tilde{N}, \mathbf{1}]$ that corresponds to Q^0 has the following property: it assigns each edge on a red snake the value 1 only if it must be 1 in every stable configuration of $[\tilde{N}, \mathbf{1}]$; likewise, each edge on a blue snake is 0 only if it must be 0 in every stable configuration of $[\tilde{N}, \mathbf{1}]$. The implication for the Stable Marriage instance is that the stable matching found by the above procedure assigns every man the best possible mate he could get in any stable matching; similarly, every woman gets the worst possible mate she could get in any stable matching. □

We may obtain the woman-optimal stable matching by an analogous procedure: eliminate the inputs of N, and then assign the remaining edges the value 1. Thus input elimination gives us both extremal matchings. The first phase of Irving's algorithm [14] also simultaneously finds both extremal matchings in instances of Stable Marriage.

6.2.3. Stable matchings are fixpoints. Any network stability problem may be viewed as a fixpoint problem: find a configuration that is a fixpoint of the network. The discussion in §6.2.2 tells us that the man-optimal and woman-optimal stable matchings of an instance of Stable Marriage are given by the least (maximally false) and greatest (maximally true) fixpoints respectively of the following system of equations: for every edge e of the comparator network, introduce a boolean variable $one(e)$ (intended to evaluate to true exactly if edge e is assigned the value 1); for each red input edge e of the network, introduce the ground condition $one(e)$; for each blue input edge e of the network, introduce the ground condition $\overline{one(e)}$; for each comparator gate with input edges i_1 and i_2 and output edges o_1 and o_2, introduce the gate equations $one(i_1) \wedge one(i_2) \Rightarrow one(o_1)$, $one(i_1) \Rightarrow one(o_2)$, and $one(i_2) \Rightarrow one(o_2)$. The system of equations is monotone because the gate equations are monotone; hence the least and greatest fixpoints are unique. Eliminating the inputs of the network corresponds to determining all those variables whose values are forced by the ground conditions. Our algorithm then computes the two extremal fixpoints by assigning all remaining variables the same value (false for the least fixpoint, true for the greatest fixpoint).

By interpreting it in the network framework, it can be seen that the proposal algorithm of [7] is a clever strategy to compute the least fixpoint of the same system of equations. It has been observed that several different serializations of the proposal algorithm yield the same final result; this is because the various serializations merely represent different routes to the unique least fixpoint.

6.2.4. Finding other stable matchings. The following lemma is useful in giving another algorithm to find a stable matching.

LEMMA 6.2. *Let N be an input-free network over a monotone scatter-free basis; let e be an edge of N. Then* $\mathrm{type}_N(e) = \{0, 1\}$.

Proof. The "all-0" and "all-1" configurations of $[N, \mathbf{e}]$ are stable; this means that e takes both values in stable configurations of $[N, \mathbf{e}]$. The result follows from Theorem 4.4. □

Let N be the comparator network corresponding to an instance of Stable Marriage; let N' be the network left after input elimination. By Lemma 6.2, each edge of N' has type $\{0, 1\}$; thus, every edge-value pair leads to a successful propagation. This means that we may, in steps 2 and 3 of Algorithm 1, pick the next edge-value pair arbitrarily and still retain the guarantee of a successful propagation; moreover, the algorithm still runs in linear time. This appears to be an interesting way to pick an arbitrary stable matching out of all the different possible stable matchings.

6.3. The lattice theorem for stable configurations of comparator networks. We show that the stable configurations of comparator networks form a distributive lattice. Conway's lattice theorem for Stable Marriage follows directly. For an introduction to the theory of lattices, see [1].

Let N be any network. The set of all configurations of N has the structure of the boolean lattice $\mathbf{2}^m$ under the natural ordering. (Here, m is the number of edges of N.) It is an easy exercise in lattice theory to show that any subset of $\mathbf{2}^m$ that is closed under the operations of bitwise *AND* and bitwise *OR* forms a distributive sublattice of $\mathbf{2}^m$ under the following definition of *meet* and *join*: the meet of two elements is their bitwise *AND* and the join of two elements is their bitwise *OR*. We show that the set of stable configurations of a comparator network is closed under bitwise *AND* and *OR*; it follows that they form a distributive sublattice of the boolean lattice of all configurations.

We need the following lemma.

LEMMA 6.3. *Let s_{in} be an input assignment to a comparator network N. Let e_1 and e_2 be two edges that are inputs to the same gate g of N. Then there cannot be two stable configurations Q_α and Q_β of $[N, s_{\mathrm{in}}]$, such that $e_1 \leftarrow 0$ and $e_2 \leftarrow 1$ in Q_α and $e_1 \leftarrow 1$ and $e_2 \leftarrow 0$ in Q_β.*

Proof. Assume the contrary, i.e., suppose $e_1 \leftarrow 0$ and $e_2 \leftarrow 1$ in Q_α and $e_1 \leftarrow 1$ and $e_2 \leftarrow 0$ in Q_β. Eliminate the inputs of N. This process will not assign a value to either e_1 or e_2, because, by hypothesis, these edges take on both possible values in the stable configurations of $[N, s_{\mathrm{in}}]$. Hence the network left after input elimination will also have two stable configurations Q'_α and Q'_β such that $e_1 \leftarrow 0$ and $e_2 \leftarrow 1$ in Q'_α and $e_1 \leftarrow 1$ and $e_2 \leftarrow 0$ in Q'_β. Hence it suffices to prove Lemma 6.3 for input-free networks. From now on, we assume $s_{\mathrm{in}} = \mathbf{e}$.

Let e_3 be the "min" output edge of g. Since $e_3 \leftarrow 0$ in Q_α and Q_β, $[e_3, 0]$ is viable in N. Hence the propagation of $[e_3, 0]$ must assign the value 0 to at least one of e_1 and e_2. Assume that e_1 is assigned the value 0. This means that e_1 is assigned the value 0 in every stable configuration of $[N, s_{\text{in}}]$ in which e_3 is assigned the value 0. This contradicts the stability of Q_β. Similarly, if e_2 is assigned the value 0, the stability of Q_α is contradicted. □

THEOREM 6.4. *Let N be a comparator network, and let s_{in} be an input assignment to N. Then the stable configurations of $[N, s_{\text{in}}]$ form a distributive sublattice of the lattice of all configurations, under the following definition of* meet *and* join: *the* meet *of two configurations is their bitwise* ***AND***; *the* join *of two configurations is their bitwise* ***OR***.

Proof. By previous remarks, it suffices to prove that the set of stable configurations is closed under bitwise ***AND*** and bitwise ***OR***.

Let Q_α and Q_β be two stable configurations of $[N, s_{\text{in}}]$. Let Q be the bitwise ***AND*** of Q_α and Q_β. The values assigned by Q_α, Q_β, and Q to any particular input edge of N are the same, so Q trivially satisfies the input conditions. Hence it suffices to show that Q satisfies the gate equations at every gate of N. Let g be an arbitrary gate of N; let e_1 and e_2 be its two input edges. The following simple case analysis shows that the gate equations are satisfied at gate g.

The first case is where one of Q_α and Q_β assigns both input edges the value 0. If this happens, the input and output edges of g are all assigned the value 0 in Q, so the gate equations are satisfied. The second case is where one of Q_α and Q_β, say Q_α, assigns both input edges the value 1. In this case, the input and output edges of g are assigned the same values in Q and in Q_β, so the gate equations are satisfied. In the third case, Q_α and Q_β both assign the same values to the two input edges. In this case, the input and output edges of g are assigned the same values in Q, in Q_α, and in Q_β, so again the gate equations are satisfied. The only remaining case is when one of Q_α and Q_β assigns the input edges the values $e_1 \leftarrow 0, e_2 \leftarrow 1$ and the other assigns the input edges the values $e_1 \leftarrow 1, e_2 \leftarrow 0$. But Lemma 6.3 assures us that this is not possible.

The case of the bitwise ***OR*** of Q_α and Q_β is analogous. □

Remarks. Tarski's theorem [28] says that the set of fixpoints of any monotone function from a complete lattice to itself forms a complete lattice. This theorem, together with Lemma 2.1, is sufficient to prove that the stable configurations (on a given input assignment) of any monotone scatter-free network form a lattice. However, this approach does not seem to yield the sublattice property. Nevertheless, and even though our proof is specific to the comparator, Theorem 6.4 does extend to any monotone scatter-free network. An easy application of the ideas in [5] yields the desired proof.

Interpreting Lemma 6.3 in the context of Stable Marriage yields the following theorem proven in [16].

LEMMA 6.5. *Let I be an instance of Stable Marriage. Suppose man m is matched to woman w in some stable matching M of I. Then there is no stable matching of I in which both m and w get better mates than in M.*

Proof. Consider the comparator network corresponding to instance I, and apply Lemma 6.3. □

An easy corollary of Theorem 6.4 is the Lattice Theorem for Stable Marriage, proven and attributed to Conway in [20].

THEOREM 6.6. *The stable matchings of a given instance of Stable Marriage form a distributive lattice, under the following definition of* meet *and* join: *The* meet *of two matchings M_1 and M_2 is a matching in which every man is married to the better of his two mates in M_1 and M_2*; *the* join *of M_1 and M_2 is a matching in which every man is married to the worse of his two mates in M_1 and M_2. (A man who is unmatched in M_1 and M_2 stays unmatched in* meet(M_1, M_2) *and* join(M_1, M_2)*).*

Remark. By Theorem 4.5, it is not possible for a man to be matched in exactly one of the two matchings.

7. Implications for parallel complexity.

7.1. Complexity theory background. For an introduction to complexity theory, see [13],[26]; for an introduction to parallel computation, see [3], [19].

The reductions in this paper are the many-one logspace reductions of [18]. We believe our reductions are simple enough that they will continue to hold under reasonable alternative definitions of reducibility. A problem is complete for a class if it is in the class and every problem in the class reduces to it. The class CC is the class of decision problems that are equivalent to Comparator Circuit Value (C-CV); see [21].

7.2. Overview of reductions. In the following sections, we show that several problems related to Stable Matching are CC-complete. Section 7.3 shows that Stable Matching and Network Stability for X-networks are essentially the same problem and that Stable Marriage and Network Stability for comparator networks are essentially the same problem. Thus, determining whether an X-network has a stable configuration is just as hard as determining whether an instance of Stable Roommates has a stable assignment; constructing a stable configuration of an X-network is just as hard as constructing a stable roommate assignment for an instance of Stable Roommates. Similar remarks apply to Stable Marriage and Network Stability for comparator networks. Of course, in this case the existence questions are trivial, because the answer is always "Yes."

Section 7.5 shows that the existence of stable configurations in X-networks is equivalent to Comparator Circuit Value. In §7.6, it is shown that constructing a stable matching for an instance of Stable Marriage is equivalent to Comparator Circuit Value: given an oracle for C-CV, we can compute the man-optimal stable matching; in the other direction, C-CV reduces to the problem of telling whether man m is married to woman w in an instance of Stable Marriage that has exactly one stable matching. This leaves the problem of constructing a stable configuration of an X-network in the nonbipartitionable case; our methods do not tell us whether this problem reduces to C-CV.

Section 7.6 also shows the following problems to be equivalent to C-CV: the Fixed Pair and Stable Pair problems, and the decision version of the Minimum-regret problem. For these problems, it does not matter whether the instance is one of Stable Matching or Stable Marriage. For the Minimum-regret problem, there is a difference between three and two: determining whether the largest regret is at most three is equivalent to C-CV, but telling whether the largest regret is two seems easier—at least it is no harder than a 2-SAT computation. (Of course, at this point, we have no proof that C-CV is harder than 2-SAT.)

Section 7.6 also shows that constructing a minimum-regret stable matching for an instance of Stable Marriage is just as hard as C-CV. The proof indicates that constructing a minimum-regret stable matching for instances of Stable Matching reduces to C-CV if and only if constructing a stable configuration of an X-network reduces to C-CV. However, we do not know how to reduce directly the construction of a minimum-regret stable matching to the construction of a stable configuration of an X-network.

Feder [5] has recently shown that (the decision problem corresponding to) constructing a stable configuration of an X-network is no harder (for parallel computation) than determining its existence. Given this theorem, it follows that all the problems considered above (except Regret-2 Stable Matching and 2-SAT) are equivalent to Comparator Circuit Value or, in other words, are CC-complete.

Finally, in §7.7, we give a reduction from C-CV to the Assignment problem (also called Bipartite Weighted Matching). This implies a fast parallel transformation of instances of Stable Marriage into instances of the Assignment problem in such a way that the solution of the Assignment problem yields the man-optimal stable matching of the Stable Marriage instance; this partially answers a question posed in [20]. However, our reduction is not entirely satisfactory, because it loses all the stable matchings except the man-optimal one. It is an open problem to give a fast parallel transformation from Stable Marriage to the Assignment problem in a manner that preserves the structure of all solutions.

7.3. Stable Matching is the same as X-Network Stability. In this section, we show that Stable Matching and X-Network Stability are essentially the same problem. There are easy reductions between the two problems that map instances of one problem to instances of the other in a manner that preserves the structure of solutions. These reductions take Stable Marriage instances into instances of Network Stability for bipartitionable X-networks. We have already seen that bipartitionable X-networks and comparator networks are interchangeable. It follows that Stable Marriage and Network Stability for comparator networks are essentially the same problem.

THEOREM 7.1. *Stable Matching is equivalent to X-Network Stability.*

Proof. The reduction from Stable Matching to X-Network Stability follows immediately from the reduction of §3. We show how to reduce X-Network Stability to Stable Matching. Given an X-network N and its input assignment s_{in}, we construct an auxiliary X-network $\tilde{N}$ in such a way that the stable configurations of $[N, s_{\text{in}}]$ are in one-to-one correspondence with the stable configurations of $[\tilde{N}, \mathbf{1}]$. The new network $\tilde{N}$ has the following properties: every snake is acyclic; no two snakes meet more than once; no snake meets itself.

The first step is to delete snakes whose input edges are assigned the value 0 by s_{in}. Observe that if an input edge of a snake is 0, every edge on this snake will take the value 0 in every stable configuration; furthermore, every snake that intersects this snake will have the same value after the intersection as it did before it. Hence, such 0-snakes may be identified and removed without altering the set of stable configurations.

The next step is to insert two *NOT*-gates in series on every noninput edge of N. We then replace each *NOT*-gate by an X-gate with one input fixed at the value 1. The resulting network is the desired network $\tilde{N}$. It is easy to check that it has the properties claimed above.

Finally, we can read off the desired instance I of Stable Matching from the network $\tilde{N}$ by inverting the reduction of §3 in the following manner. Introduce a person for each snake of $\tilde{N}$. Two persons find each other acceptable if and only if their snakes meet. Finally, the order in which a person's snake meets the other snakes gives his preference ordering. It follows from Theorem 3.2 that the stable configurations of $[N, s_{\text{in}}]$ are in one-to-one correspondence with the stable matchings of instance I. □

Remarks. The above proof actually proves a little more. First, if N is bipartitionable, so is $\tilde{N}$, and then instance I will be an instance of Stable Marriage. Second, the sizes of $\tilde{N}$ and I are linear in the size of N. Finally, note that in $\tilde{N}$, each snake meets at most three other snakes, and furthermore, if a snake meets more than one other snake, the value on its output edge is 0. This means that in instance I, each person has at most three persons on his preference list. Also, in every stable matching of I, the persons who list more than one other person are guaranteed to be matched to one of their first three choices. We shall use this later, to prove Theorem 7.15.

We give an easy proof, based on the network formulation of Stable Matching, of the following well-known lemma.

LEMMA 7.2. *Complete Stable Matching is no easier than Stable Matching. Complete Stable Marriage is no easier than Stable Marriage.*

Proof. Given an instance of Stable Matching, we convert it to an instance of Complete Stable Matching, without affecting the set of stable matchings. The first step is to introduce the appropriate number of extra persons and decree that they have no acceptable mates. This does not affect the set of stable matchings. In terms of X-networks, introducing these extra persons amounts to adding new snakes with input 1 as separate connected components.

It is instructive to do the rest of the proof with networks instead of preference lists. Let N_1 be the X-network corresponding to the Stable Matching instance we have so far. Network N_1 has one deficiency: each snake of N_1 might not meet every other snake that it should meet. We remedy this by a "cascade" construction, i.e., by appending a new network N_2 at the "back" of N_1. The outputs of N_1 become the inputs of N_2. Our final network N will be the composition of these two networks.

Let us examine what properties N_2 should have. First, each snake of N_2 must meet a prescribed set of the other snakes, each exactly once. Second, the sets of stable configurations of $[N, \mathbf{1}]$ and $[N_1, \mathbf{1}]$ should be isomorphic. The second condition will be met if the following condition is met: whenever s is the output word of some stable configuration of $[N_1, \mathbf{1}]$, $[N_2, s]$ has exactly one stable configuration. A simple solution is to make N_2 a circuit. This works because every circuit has exactly one stable configuration on any given input assignment.

We construct N_2 as follows. Pick any fixed total order O of the snakes, and let each snake meet all the snakes it ought to meet in the order induced by O. This defines N_2 and hence N. The desired instance of Stable Matching can be read off from N. This completes the reduction. The proof for Marriage instances is similar.

Notice that the cascade construction actually does something very simple with the preference lists: it enlarges the preference list of each person x, by appending to it all the mates originally unacceptable to x, in the order given by O. □

7.4. On reducing problems to Comparator Circuit Value. Following [18], we say that a problem Z is *many-one* reducible to C-CV if we can exhibit a reduction from Z to C-CV that takes each instance of Z to a comparator circuit plus an input assignment to it, so that the instance has a "Yes" answer if and only if the circuit has output 1 when evaluated on the given input assignment. In proofs, it is more convenient to think in terms of Turing reductions to C-CV. The following lemma helps convert Turing reductions to many-one reductions.

LEMMA 7.3. *If* Z_1 *and* Z_2 *are many-one reducible to* C-CV, *so are* $\mathsf{OR}(Z_1,Z_2)$, $\mathsf{AND}(Z_1,Z_2)$, *and* $\mathsf{NOT}(Z_1)$.

Proof. The many-one reducibility of $\mathsf{OR}(Z_1,Z_2)$ and $\mathsf{AND}(Z_1,Z_2)$ to C-CV follows from the fact that the comparator can simulate the OR-gate and the AND-gate. The many-one reducibility of $\mathsf{NOT}(Z_1)$ to C-CV follows because the X-gate can simulate $\{\mathrm{C}, \mathsf{NOT}\}$ and because X-CV reduces to C-CV by Lemma 7.4. □

LEMMA 7.4. X-CV *is equivalent to* C-CV.

Proof. The basis $\{X\}$ can simulate $\{C\}$; hence C-CV reduces to X-CV. The reduction in the other direction utilizes Goldschlager's idea of using "double-rail" logic [9]. In other words, encode the values 0 and 1 by the pairs $(1, 0)$ and $(0, 1)$ respectively. Replace every edge e in the original circuit by a pair of edges (e^-, e^+); the rest of the construction will guarantee that the two edges in a pair will always have complementary values. If e is an input edge that is assigned the value v, edges e^- and e^+ are assigned the values $\overline{v}$ and v respectively. Finally, we must replace every X-gate by an appropriate gate. The X-gate has two inputs i_1, i_2 and two outputs $o_1 = i_1 \overline{i_2}$, $o_2 = \overline{i_1} i_2$. The new gate will have four inputs $i_1^-, i_1^+, i_2^-, i_2^+$ and four outputs $o_1^-, o_1^+, o_2^-, o_2^+$. The new gate must simulate the behaviour of the X-gate in the following sense: when presented with an input word that is a valid encoding of the input word to an X-gate, it must produce an output word that is the valid encoding of the corresponding output word of the X-gate. The action of the gate on input words like 0010 may be arbitrarily chosen. One solution is to use the gate defined by the equations $o_1^- = i_1^- + i_2^+$, $o_1^+ = i_1^+ i_2^-$, $o_2^- = i_1^+ + i_2^-$, $o_2^+ = i_1^- i_2^+$. It is easily seen that this gate can be simulated by the comparator.

An easy inductive proof shows that if edge e in the original circuit has value v, then the pair of edges (e^-, e^+) in the new circuit will be assigned the value-pair $(\overline{v}, v)$. Hence, in particular, if e_0 is the output edge of the original circuit, the edge e_0^+ will serve as the output edge of the new circuit. Finally, since the new gate can be simulated by the comparator, the new circuit can be converted into a comparator circuit. □

7.5. X-Network Stability is equivalent to Comparator Circuit Value. Let Ω be a scatter-free basis that can simulate $\{NAND\}$. The objective of this section is to show that Ω-NS and Ω-CV are equivalent. (This result was presented in [21]; here we give a complete proof.) In particular, X-NS is equivalent to X-CV; hence, using Lemma 7.4, X-NS is equivalent to C-CV.

7.5.1. Reducing Ω-CV to Ω-NS. The proof uses the idea of a *forcer*. A v-forcer is a one-input, zero-output network that has a stable configuration if and only if the input takes the value v. A 0-forcer may be built by tying the output of a $NAND$-gate to one of its inputs, and a 1-forcer may be obtained by adding a NOT-gate at the input of a 0-forcer.

LEMMA 7.5. *Let Ω be any basis that can simulate $\{NAND\}$. Then Ω-*CV *reduces to Ω-NS.*

Proof. To determine the output of a circuit over Ω, feed its output into a 1-forcer. The resulting network has a stable configuration if and only if the circuit has output 1. Since basis Ω can simulate $\{NAND\}$, we can build forcers over Ω. □

7.5.2. Reducing Ω-NS to Ω-CV. Given a network N over a scatter-free basis Ω and an input assignment s_{in} to N, we wish to construct a circuit over Ω so that the output of the circuit is 1 if and only if $[N, s_{\text{in}}]$ has a stable configuration.

Let N' be the network left after the inputs of N are eliminated. We say that an edge of N is *bad* if it is a contradicting edge of N'. By Theorem 4.3, network N has no stable configuration if and only if it has a bad edge. Thus the key step is to construct a circuit that will tell us if a given edge is bad. Once this is done (independently) for every edge of N, we may combine the results and answer the stability question by using some simple additional circuitry. The rest of the discussion is divided into three stages. In the first stage, we show how to tell if an edge is bad; in the second stage, we define a network process that will compute whether or not an edge is bad; in the final stage, we convert the network process into a circuit computation.

Stage 1. How do we tell if edge e is bad? Break edge e, thus creating an extra input e(in) and an extra output e(out). Call the resulting network N_e. Let s_e be the input assignment to N_e obtained by augmenting s_{in} with the assignment $e(\text{in}) \leftarrow v$. Consider the following experiment Expt(e, v): Assign each input of N_e the value it gets under the input assignment s_e, and perform input elimination. Since the gates are scatter-free, each output of N_e will be assigned a value. We say that experiment Expt(e, v) succeeds if the value assigned to edge e(out) is v; otherwise it fails.

LEMMA 7.6. *Edge e is bad if and only if both the experiments* Expt(e, 0) *and* Expt(e, 1) *fail.*

Proof. The key observation is that the result of input elimination does not depend on the order in which the inputs are eliminated. Hence we may pretend that experiment Expt(e, v) is conducted in two phases: in the first phase, all the inputs except e(in) are eliminated; in the second phase, input e(in) is eliminated. There are two cases, depending on whether or not edge e(out) is assigned a value in the first phase.

Case 1. Suppose edge e(out) is assigned a value v' in the first phase. This means that edge e would be assigned the value v' when the inputs of N are eliminated. It follows that e is not an edge of N'; hence e is not a bad edge of N. Notice also that Expt(e, v') will succeed. Thus Lemma 7.6 is true in this case.

Case 2. Suppose edge e(out) is not assigned a value in the first phase. This means two things: first, the effect of the first phase is identical to that obtained by eliminating the inputs of N; second, edge e is an edge of N'. Notice that the second phase of Expt(e, v) will assign value v to edge e(out) if and only if $[e, v]$ is viable in N'. It follows that both Expt$(e, 0)$ and Expt$(e, 1)$ fail if and only if edge e is contradicting in N'. Hence Lemma 7.6 is true in this case as well. □

Stage 2. We now define a network process that will compute the value assigned to edge e(out) by experiment Expt(e, v). Consider the following network process: Initialize network N_e to any configuration consistent with the input assignment s_e. Then let the network "run." In other words, perform the following primitive operation at each gate of the network, in parallel: apply the gate function to the values on the input edges, and use the result to update the values on the output edges of the gate. The crucial observation is that the values on certain edges of the network will stabilize, i.e., will not change after a certain number of parallel steps. In particular, the value on edge e(out) will stabilize at the value it is assigned in experiment Expt(e, v).

LEMMA 7.7. *Each edge that is assigned a value in experiment* Expt(e, v) *will stabilize at the same value in the network process.*

Proof. Notice that the value on each input edge of network N_e is held constant. Hence Lemma 7.7 is true for these edges. The process of eliminating the inputs of N_e in experiment Expt(e, v) gradually assigns values to some edges of N_e. An assignment to an output edge of gate g follows from an assignment to a subset of the inputs of g by invoking the equation of g. This process is faithfully mimicked in the network process: first each edge in the relevant subset of the inputs stabilizes to the correct value; then the output edge stabilizes. □

The proof of Lemma 7.7 may be extended to bound the number of parallel steps needed for edge e(out) to stabilize. Let m be the number of edges in network N. Then the longest chain of influence in the input elimination process of experiment Expt(e, v) involves at most $m + 1$ edges. Hence edge e(out) will reach its correct value after at most m parallel steps.

Stage 3. The final step is to use the regularity of the network process to "unravel" it into a circuit computation. In other words, we construct a circuit over Ω that will simulate the network process. The idea is to make m copies of network N_e (we call each copy a layer) and connect these layers up into a circuit C in such a way that the values that leave the tth layer are the values that appear in network N_e after parallel step t. As a result, the correct value of edge e(out) may be read off from the top layer of the circuit.

More accurately, we make m copies of network N_e and change the output edges of all the gates so that they enter the next layer, instead of the same layer. The gates of C are $\{(g, t) \mid g$ is a gate of N_e; $1 \le t \le m\}$; the edges are $\{(\tilde{e}, t) \mid \tilde{e}$ is an edge of N_e; $0 \le t \le m\}$. (Not quite! The edges of the form $(\tilde{e}, 0)$ for $\tilde{e}$ an output edge of N_e or of the form $(\tilde{e}, m)$ for $\tilde{e}$ an input edge of N_e are not actually present.) The gates $\{(g, t) \mid 1 \le t \le m\}$ all have the same gate type as g. If g has inputs $i_1, \ldots, i_\lambda$ and outputs $o_1, \ldots, o_\mu$, then (g, t) has inputs $(i_1, t-1), \ldots, (i_\lambda, t-1)$ and outputs $(o_1, t), \ldots, (o_\mu, t)$.

If $\tilde{e}$ is an input edge of N_e with value $\tilde{v}$, all the input edges $\{(\tilde{e}, t) \mid 0 \le t < m\}$ get the value $\tilde{v}$ in C; this has the effect of presenting the real inputs of N_e to each layer of the circuit. The edges $(\tilde{e}, 0)$, for $\tilde{e}$ an internal edge of N_e, are assigned arbitrary values; this reflects the arbitrariness in the initial configuration of N_e. Finally, if $\tilde{e}$ is a noninput edge of N_e, edge $(\tilde{e}, m)$ is an output edge of C. This completes the description of C.

The following lemma may be proven by an easy induction on t.

LEMMA 7.8. *The value on edge $(\tilde{e}, t)$ of C equals the value on edge $\tilde{e}$ of N_e after parallel step t.*

In particular, the value assigned to edge e(out) in experiment Expt(e, v) is given by the value assigned to edge $(e(\text{out}), m)$ in C. Once we have the values assigned to edge e(out) in experiments Expt$(e, 0)$ and Expt$(e, 1)$, we may determine whether or not edge e is bad by adding an *AND*-gate and a *NOT*-gate. We may then combine the results for each edge and answer the stability question by adding *OR*-gates and a *NOT*-gate. Thus we get a circuit over $\Omega\cup\{NAND\}$ that computes the answer to the question "Does $[N, s_{\text{in}}]$ have a stable configuration?" This circuit has $O(m^3)$ gates.

THEOREM 7.9. *Let Ω be any scatter-free basis that can simulate $\{NAND\}$. Then Ω-NS is equivalent to Ω-CV.*

Proof. The reduction from Ω-NS to Ω-CV follows from the construction above; the reduction from Ω-CV to Ω-NS follows from Lemma 7.5. □

7.6. More Stable Matching problems equivalent to Comparator Circuit Value. In this section, we prove that several Stable Matching problems are CC-complete. We begin by showing (Lemma 7.10) that X-network stability problems remain just as hard even if we place certain constraints on the values that certain edges can take.

LEMMA 7.10. *The following problems are equivalent (and hence equivalent to* C-CV):

1. *Given an X-network N, an input assignment s_{in}, and a partial assignment P to the noninput edges of N, can P be completed to a stable configuration of the network?*
2. *Given an X-network N and an assignment s_{in} to the input edges of N, is there a stable configuration of $[N, s_{\text{in}}]$?*

Proof. Problem 2 is clearly a special case of problem 1, so it suffices to reduce problem 1 to problem 2. This can be done by using forcers. Suppose P requires that edge e get the value v. Break edge e, creating input e(in) and output e(out). Assign edge e(in) the value v, and feed e(out) into a v-forcer. □

THEOREM 7.11. *The following problems reduce to* C-CV.

1. *Fixed Pair. Is the given pair of persons a fixed pair of the given instance of Stable Matching, i.e., is it true that these two persons are matched to each other in every stable matching?*
2. *Stable Pair. Is the given pair of persons a stable pair, i.e., is it true that there is a stable matching that matches these two people to each other?*
3. *Minimum-regret. Is it true that there is a stable matching of the given instance in which every person has regret at most k?*

Proof. To check that two persons form a fixed pair, it suffices to check that the snakes corresponding to these two persons always meet each other with the value 1 in the corresponding instance of X-Network Stability. Let g be the gate where the two snakes meet; we have to check that there is no stable configuration that assigns either edge entering g the value 0. This reduces to C-CV by Lemma 7.10 and Lemma 7.3.

To check that two persons form a stable pair, it suffices to check that there is a stable configuration that matches these two persons to each other, i.e., a stable configuration in which the snakes corresponding to these two people meet each other with the value 1. This reduces to C-CV by Lemma 7.10.

To check that there is a stable matching in which every person has regret at most k, just check whether there is a stable configuration that assigns each snake the value 0 after it has traversed k gates. (If snake q has fewer than k gates on it, the desired stable configuration must assign the output edge of q the value 0.) Again, we invoke Lemma 7.10 to reduce this problem to C-CV. □

THEOREM 7.12. *The Man-optimal Stable Marriage problem, "Given an instance I of Stable Marriage, a man m, and a woman w, determine whether m is matched to w in the man-optimal stable matching M^0 of I," reduces to* C-CV.

Proof. Let $[N, s_{\text{in}}]$ be the instance of Comparator Network Stability corresponding to instance I; let Q^0 be the stable configuration of $[N, s_{\text{in}}]$ corresponding to the man-optimal stable matching of I. We show how to construct a comparator circuit to compute Q^0. The proof of Theorem 7.12 then follows by applying Lemma 7.3.

Theorem 6.1 tells us that Q^0 is the "most-0" stable configuration of $[N, s_{\text{in}}]$. We claim the following network process finds Q^0. Initialize N to the "all-0" stable configuration. Then make all the red inputs 1. These changes create disturbances that propagate through the network. But the network will eventually enter the stable configuration Q^0.

We justify the claim. Notice that all the input disturbances "increase" the value assigned to an input edge. The comparator is a monotone gate—increasing the values assigned to the inputs can never cause an output to decrease. Hence, the effect of the disturbances as they move through the network will be to increase the value of some edges of N, but no edge will ever move from 1 to 0. It follows that the disturbances will die out in at most m steps. Now observe that an edge is assigned the value 1 during the network process only if it must have the value 1 in every stable configuration of $[N, s_{\text{in}}]$. It follows that the stable configuration the network reaches is Q^0.

The network computation on N may be unravelled into a circuit computation over the same basis in the standard fashion. (See, for example, the discussion in §7.5.) □

We can also reduce the problem of constructing a minimum-regret stable matching for an instance of Stable Marriage to C-CV. Let I be any instance of Stable Marriage; let regret(I) be the value of the largest regret in a minimum-regret stable matching of I. The discussion in §4.2 tells us that there is a single instance $I_{\text{regret}(I)}$ of Stable Marriage whose stable matchings are all the minimum-regret stable matchings of I. Define the man-optimal minimum-regret stable matching of I to be the man-optimal stable matching of $I_{\text{regret}(I)}$. The following theorem shows that the construction of this special stable matching reduces to C-CV.

THEOREM 7.13. *The Man-optimal Minimum-regret Stable Marriage problem, "Given an instance I of Stable Marriage, a man m and a woman w, determine whether m is matched to w in the man-optimal minimum-regret stable matching of I," reduces to* C-CV.

Proof. Assume without loss of generality that there are at least as many men as women in instance I; let n be the number of men. Then regret(I) is one of the integers between 1 and $(n + 1)$. If regret$(I) = (n + 1)$, this is taken to mean that there is some person who is unmatched in every stable matching of I. The idea is to compute the man-optimal stable matching separately in each of the $(n + 1)$ cases and then select the correct stable matching using the value of regret(I).

Suppose we wish to compute whether m is matched to w in the matching that is best for the men from among all the stable matchings in which each person has regret at most k. First compute the appropriate instance I_k of Stable Marriage as in §4.2. The proof of Theorem 7.12 can be made to yield a comparator circuit C_k whose output is 1 if and only if man m is matched to woman w in the man-optimal stable matching of instance I_k.

The reduction from the Minimum-regret problem to C-CV (see Theorem 7.11 above) can be made to yield, for each $k \in [1, (n + 1)]$, a comparator circuit Atmost$_k$ whose output is 1 if and only if regret$(I) \leq k$. (The circuit Atmost$_{n+1}$ is trivial; its output is always 1.) Now build a comparator circuit Exact$_k$, where the output of Exact$_k$ is 1 if and only if regret$(I) = k$. For $k > 1$, regret$(I) = k$ if and only if Atmost$_k$ has output 1 but Atmost$_{k-1}$ has output 0; Lemma 7.3 then yields the comparator circuit Exact$_k$. Also, Exact$_1$ is just Atmost$_1$. (Of course, for these constructions, we need two copies of the circuits Atmost$_k$.)

The circuit $\underset{k}{\mathit{OR}}(\mathit{AND}(\text{Exact}_k, C_k))$ has output 1 if and only if man m is matched to woman w in the man-optimal minimum-regret stable matching of I. This circuit can be built with comparators—just simulate the AND-gate and the OR-gate with comparators. □

The problems reduced to C-CV in the previous three theorems are in fact equivalent to C-CV.

THEOREM 7.14. C-CV *reduces to the Fixed Pair problem, the Stable Pair problem, the Man-optimal Stable Marriage problem, and the Man-optimal Minimum-regret Stable Marriage problem.*

Proof. Let C be the given comparator circuit. Use the transformation of Fig. 2(b) to replace it by a bipartitionable X-circuit. Delete snakes whose input edges have the value 0, as in the proof of Theorem 7.1. Circuit C has output 1 (on a given input assignment s_{in}) if and only if the output edge of a certain vertical snake q of the X-circuit is assigned the value 1. Add a new horizontal snake q' with input 1 to the X-circuit. Let q' meet q and no other snake. Let q meet q' last. Then C has output 1 on s_{in} if and only if q and q' meet each other with value 1 in the new X-circuit. The method used in the proof of Lemma 7.2 can be used to ensure that the number of horizontal and vertical snakes are equal and that every horizontal snake meets every vertical snake exactly once. Now, read off an instance of Complete Stable Marriage from the X-gate circuit. This instance will have exactly one stable matching. Also, the output of C is 1 on s_{in} if and only if the persons corresponding to snakes q and q' are married to each other in the unique stable matching. The theorem follows. □

THEOREM 7.15. C-CV *reduces to the question "Does the given instance of Complete Stable Marriage have a stable matching in which every person has regret at most* 3?"

Proof. Given an instance of C-CV, we produce an instance of Complete Stable Marriage with the following properties: there is exactly one stable matching; the regret of every person except woman w in this stable matching is at most 3; woman w has regret 3 if and only if the instance of C-CV has output 1, otherwise she has regret 4.

As in the proof of Theorem 7.14, replace the given comparator circuit C by a bipartitionable X-circuit without 0-snakes, so that C has output 1 on input assignment s_{in} if and only if the output edge of a certain vertical snake q_1 of the X-circuit is assigned the value 1.

The next step is to prepare the snake of woman w appropriately. Introduce the following new snakes: $q_0, q_2, q_3, q_4, q_5, q_7$. The input edge of each of these snakes is assigned the value 1. The snakes with odd subscripts will be vertical snakes, and the snakes with even subscripts will be horizontal snakes. Snake q_0 will ultimately become the snake of woman w. Snake q_0 meets q_3, q_5, q_1, q_7 in that order; snake q_2 meets q_3 only; snake q_3 meets q_2 and q_0 in that order; snake q_4 meets q_5 only; snake q_5 meets q_4 and q_0 in that order; snake q_7 meets q_0 only. Snake q_1 meets q_0 last. The effect of this construction is to guarantee that woman w gets either her third choice or her fourth choice; she gets her third choice if and only if circuit C on input s_{in} has value 1.

The rest of the proof contains a general construction that can be used to simultaneously reduce the regret of some or all of the participants to three and make the instance "complete," without changing the structure of the set of all stable matchings. It introduces new persons, but each of them has regret at most three. The same construction works, with the appropriate changes, for the nonbipartitionable case. The size of the output instance produced by the construction is $O(n^4)$, where n is the size of the comparator circuit; it is possible to produce an instance of size $O(n^2)$ if the output instance is not required to be bipartitionable, but such considerations are largely irrelevant from the viewpoint of parallel reductions.

The method used in the proof of Lemma 7.2 converts our bipartitionable X-network into the X-network corresponding to an instance of Complete Stable Marriage. There is one difficulty, however: some of the participants other than w might have regret more than 3, or in other words, some snakes other than q_0 might have the value 1 even after meeting three other snakes. This can be rectified by performing some minor surgery on the network. The trick, much as in the proof of Theorem 7.1, is to insert two *NOT*-gates in series into every noninput edge of the network except the edges on snake q_0 and then to replace each *NOT*-gate by an X-gate with input 1. The result is a bipartitionable X-circuit with the following properties: each snake except q_0 meets at most three other snakes; the value on the input edge of every snake is 1; the value on the output edge of every snake is 0; there are an equal number of man-snakes and woman-snakes. Now convert this to an instance of Complete Stable Marriage by completing each person's list in any manner. The resulting instance will have all the desired properties. □

It follows from the following theorem and Theorem 4.5 that the question "Does the given instance of Stable Matching have a stable matching in which every person has regret at most 2?" is in NC. This suggests that the number 3 in Theorem 7.15 cannot be reduced.

THEOREM 7.16. *Stable Matching is in* NC *if each person lists at most two other participants as acceptable mates.*

Proof. Introduce for each person x two boolean variables $\text{gets}(x, 1)$ and $\text{gets}(x, 2)$. The intended meaning is that $\text{gets}(x, j)$ is made true by a stable matching if and only if x is matched to his jth choice. Then the stable matchings are precisely the solutions of the following equations.

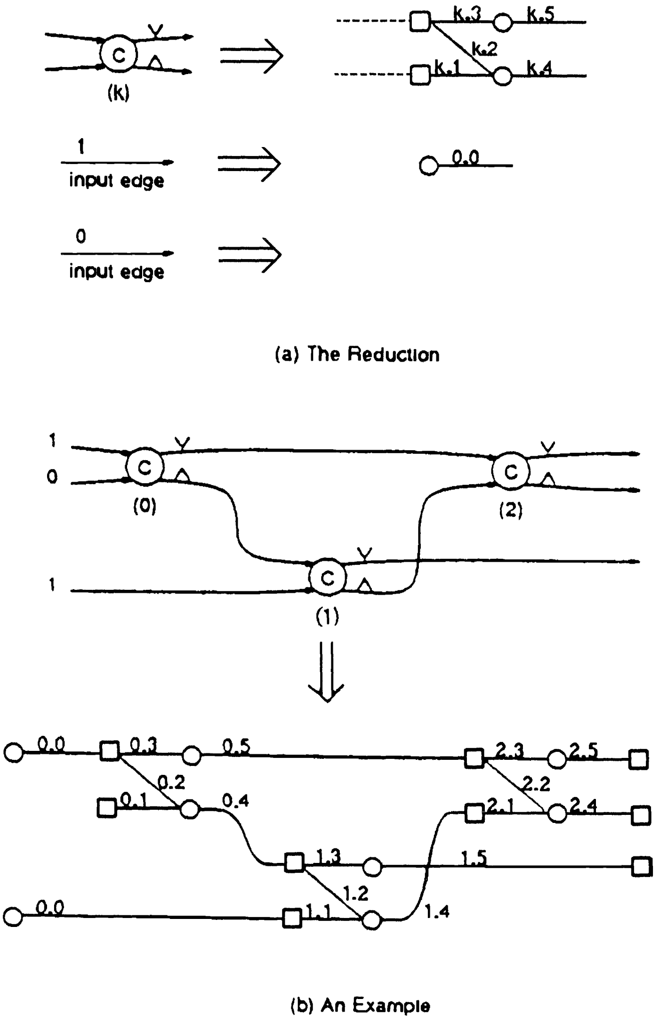
1. $\overline{\text{gets}(x, 1) \wedge \text{gets}(x, 2)}$.
2. If x is the jth choice of y and y is the kth choice of x, then $\text{gets}(y, j) \equiv \text{gets}(x, k)$.
3. If x is the first choice of y and y is the first choice of x, then $\text{gets}(x, 1)$.
4. If x is the first choice of y and y is the second choice of x, then $\text{gets}(x, 1) \vee \text{gets}(x, 2)$.
5. If x is the second choice of y and y is the second choice of x, then $\text{gets}(x, 1) \vee \text{gets}(y, 1) \vee \text{gets}(x, 2)$.

Notice that if x and y are second choices of each other, then all the equations involving $\text{gets}(x, 2)$ and $\text{gets}(y, 2)$ may be replaced by the single defining equation $\text{gets}(x, 2) \equiv \text{gets}(y, 2) \equiv \overline{\text{gets}(x, 1) \vee \text{gets}(y, 1)}$. We say that such variables $\text{gets}(x, 2)$ and $\text{gets}(y, 2)$ are *inessential.* Solve the original system of equations as follows: delete all the equations involving inessential variables; solve the resulting system of equations; and then solve for the inessential variables by using their defining equations. The key observation is that eliminating the inessential variables renders the system of equations simpler—each equation in the new system mentions at most two variables. Hence the new system may be converted into an instance of 2-SAT and can be solved in NC [4]. □

Remark. The difference between preference lists of lengths two and three has also been observed by Soroker [25]. He considers a special case of Stable Marriage, called Priority Marriage. He shows that Priority Marriage is just as hard even if each preference list is at most three long; however, if each list has at most two persons on it, the problem is in NC.

7.7. A reduction from Comparator Circuit Value to the Assignment problem. In this section, we exhibit a fast parallel reduction from C-CV to Bipartite Weighted Matching. Cook [2] initially showed that X-CV is equivalent to a problem called Lex-first Maximal Matching; also, there are easy reductions from C-CV to X-CV and from Lex-first Maximal Matching to Weighted Matching. The actual reduction used in this section incorporates these ideas; the specific gadgets used are due to Soroker [25].

Let C be the given comparator circuit, with n gates; let s_{in} be the given input assignment to C. We shall assume that C is topologically sorted, that is, we are given a function *num* that maps the gates of C into distinct integers in $[0, \ldots, (n - 1)]$, in such a way that if an output edge of g_1 is an input edge of g_2, then $\text{num}(g_1) < \text{num}(g_2)$. This assumption is justified (see [24]) because there is a fast parallel transformation that takes as input any circuit C and an input assignment s_{in} and returns a (larger) circuit C' over the same basis, a topological-sorting function num for C', and an input assignment s'_{in}, so that $[C', s'_{\text{in}}]$ has output 1 if and only if $[C, s_{\text{in}}]$ has output 1. We shall also assume, without loss of generality, that the output whose value we wish to compute is the *OR* output of gate $(n - 1)$. We shall construct a bipartite graph G and a function *weight* from the edges of G into the positive integers, so that the maximum-weight matching of G is unique; furthermore, this matching has odd weight if and only if the output of C on the input assignment s_{in} is 1.



(a) The Reduction

(b) An Example

FIG. 3. *Reducing Comparator Circuit Value to Bipartite Weighted Matching.*

The construction is illustrated in Fig. 3(a). Each comparator gate g is locally transformed into a graph gadget with four vertices and five edges. If $\text{num}(g) = k$, the edges of the gadget are labeled from $k.1$ to $k.5$ as shown. The weight of an edge labeled $k.l$ will be $2^{5n-(5k+l)}$. Think of the gadget as taking in two input edges (shown dotted) and producing two output edges, the edges labeled $k.4$ and $k.5$. The gadgets for different gates are connected to one another in exactly the same way as the gates are connected in C.

Input edges of C are treated as follows. Suppose e is an input edge of C. If e is assigned the value 1 by s_{in}, it becomes an edge with label 0.0 (and weight 2^{5n}) in G. On the other hand, if e is assigned the value 0, it does not appear in G. An example of the transformation is shown in Fig. 3(b).

It is easy to prove that G is bipartite—the shape of the vertices (square or round) in Fig. 3 gives a bipartition.

THEOREM 7.17. *The maximum-weight matching of G is unique, and it has odd weight if and only if the value assigned to the output edge of C on input assignment s_{in} is* 1.

Proof. Say an edge is *heavy* if its weight is 2^{5n}; it is *light* otherwise. Every heavy edge will appear in every maximum-weight matching of G, because these edges do not share vertices, and the combined weight of all the light edges is $2^{5n} - 1$, which is less than 2^{5n}. The weights of the light edges are decreasing powers of 2, and no two light edges have the same weight. This means that the heaviest light edge weighs more than all the other light edges combined, and hence it is always advantageous to add the heaviest light edge to the current matching. Reasoning in this fashion, we conclude that the maximum-weight matching is unique and can be found by the following greedy algorithm. Start with a matching consisting of the heavy edges, and then inspect each light edge in decreasing order of weight, adding it to the current matching if at all possible.

We prove the following inductive assertion.

($*$) Edge e of C is assigned the value 1 in the unique stable configuration of $[C, s_{\text{in}}]$ if and only if the corresponding edge in G is matched in the unique maximum weight matching M of G.

We first prove ($*$) when e is an input edge. Then we consider each gate of G, in increasing order of $\text{num}(g)$, and prove that if ($*$) is true for the edges that are inputs to g, it must also be true for the edges that are outputs of g. This will complete the proof of ($*$).

The base case is when e is an input edge of C. If e is assigned the value 0 by s_{in}, ($*$) is vacuously true. If e is assigned the value 1 by s_{in}, the corresponding edge in G is heavy and will be included in M.

Now assume that ($*$) is true for the edges that are inputs to the gate g with $\text{num}(g) = k$. We wish to prove ($*$) for the outputs of g. To do this, examine the graph gadget corresponding to gate g. See Fig. 3(a). Define the boolean variables a_1 and a_2 as follows: a_1 is true if and only if the top input to g is 1; a_2 is true if and only if the bottom input to g is 1. By assumption, these variables also capture whether or not the input edges to the graph gadget are matched in M. Now notice that in the vicinity of the graph gadget, the weights decrease in the following order: the input edges (in some order); then the edges labeled $k.1$, $k.2$, $k.3$, $k.4$, $k.5$, in this order; then the other edges, if any, that are incident to the right endpoints of the edges labeled $k.4$ and $k.5$. This allows us to infer when (under what conditions on the variables a_1 and a_2) the greedy algorithm will match the edges labeled $k.l$. The computations in Table 2 verify that ($*$) holds for the output edges of g.

TABLE 2
When the edges of the graph gadget are matched.

Edge	When Matched
Top input edge	a_1
Bottom input edge	a_2
Edge labeled $k.1$	$\overline{a_2}$
Edge labeled $k.2$	$\overline{a_1}\, a_2$
Edge labeled $k.3$	$\overline{a_1}\, \overline{a_2}$
Edge labeled $k.4$	$a_1\, a_2$
Edge labeled $k.5$	$a_1 + a_2$

Given ($*$), we complete the proof of Theorem 7.17. All the edges of G have even weights except the edge labeled $(n - 1).5$, which has weight 1. Hence M has odd weight if and only if this edge is in M, which happens if and only if the output edge of C is assigned the value 1 when C is evaluated on input assignment s_{in}. □

Remark. By combining the reduction of this section with the reduction in the proof of Theorem 7.12, we get a reduction from the Man-optimal Stable Marriage problem to Bipartite

Weighted Matching. This reduction takes an instance of Stable Marriage with n men and n women to an instance of Bipartite Weighted Matching that has $(2+o(1))n^4$ vertices on either side of the graph and edge weights that have $(5+o(1))n^4$ bits each. Given the optimal matching M of the Weighted Matching instance, we can read off the man-optimal stable matching of the Stable Marriage instance.

8. Other reductions.

8.1. Counting the number of stable matchings is #P-complete. We show how to use the network formulation to give a simple proof that the problem of counting the number of stable matchings in an instance of Stable Matching is #P-complete. A stronger result is known [16], namely, that the theorem holds even if the instance is one of Stable Marriage, but the proof of the stronger result uses a nontrivial theorem about the structure of all stable matchings of an instance of Stable Marriage.

THEOREM 8.1. *The problem of counting the number of stable configurations in an instance of X-Network Stability is #P-complete.*

Proof. Testing whether a configuration of an X-network is stable is clearly in P; this puts the counting problem in #P. To show completeness for #P, we give a parsimonious reduction from Monotone 2-SAT. Monotone 2-SAT is shown to be #P-complete in [29].

Let I be the instance of Monotone 2-SAT. Construct X-network N as follows: it has a cyclic snake q_a for each variable a of I; snakes q_a and q_b intersect if and only if the clause $a \vee b$ appears in I. The instance of X-Network Stability is $[N, \mathbf{e}]$.

Notice that all the edges of any snake q_a are assigned the same value in any stable configuration of $[N, \mathbf{e}]$. If this value is 0, this is taken to mean that the variable a is "true." It is easy to check that this gives a one-to-one correspondence between the stable configurations of $[N, \mathbf{e}]$ and the satisfying assignments of I. □

8.2. Three-party Stable Marriage is NP-complete. Knuth [20] asks about the complexity of Stable Marriage if the participants are of three kinds: men, women, and dogs. We show that one possible formulation of this problem is NP-complete. It has been independently shown in [22] that this problem is NP-complete.

We consider the following formulation. An instance of Three-party Stable Marriage has equal numbers of three kinds of participants, say men, women, and dogs. A matching M is a set of triplets: each triplet contains exactly one participant of each kind, and each participant is in exactly one triplet of M. The participants have preference lists. Each participant lists all the triplets containing his name in order of preference. Matching M is unstable if there is a man m, a woman w, and a dog d, each of whom prefers the triplet (m, w, d) to the triplet he is assigned to in M. The problem is: given an instance of Three-party Stable Marriage, determine whether there is a stable matching.

LEMMA 8.2. *Three-party Stable Marriage is equivalent to Y-Network Stability.*

Proof. Section 3 shows that every instance of Stable Matching is an instance of X-Network Stability. In an analogous fashion, it can be shown that every instance I of Three-party Stable Marriage is an instance $[N, \mathbf{1}]$ of Y-Network Stability. The Y-gate is a natural generalization of the X-gate. It has three inputs i_1, i_2, i_3 and three outputs o_1, o_2, o_3. The function $Y : i_1, i_2, i_3 \rightarrow o_1, o_2, o_3$ is defined as follows: $Y(1,1,1) = (0,0,0)$; on all other input words, $Y(i_1, i_2, i_3) = (i_1, i_2, i_3)$. Input i_j and output o_j are associated. This association allows us to describe Y-networks in terms of snakes, just as we did for X-networks.

Network N contains one acyclic snake per participant. Snakes meet in triples at Y-gates. The sequence of Y-gates on the snake of a participant corresponds exactly to the sequence of triples that appear on his preference list. Proceeding in much the same way as in §3, it can be shown that snakes drop in triples in any stable configuration of $[N, \mathbf{1}]$, and that the stable matchings of I are in one-to-one correspondence with the stable configurations of $[N, \mathbf{1}]$—three persons are matched in I if and only if their snakes drop together in $[N, \mathbf{1}]$. This completes the reduction from Three-party Stable Marriage to Y-Network Stability.

The reduction from Y-Network Stability to Three-party Stable Marriage is similar to the reductions in §7.3. Given a Y-network N and its input assignment s_{in}, we construct an auxiliary Y-network $\tilde{N}$ in such a way that the stable configurations of $[N, s_{\text{in}}]$ are in one-to-one correspondence with the stable configurations of $[\tilde{N}, \mathbf{1}]$; furthermore, network $\tilde{N}$ has the following properties: every snake is acyclic; no two snakes meet more than once; no snake meets itself; each snake is assigned a *color* $\in \{1, 2, 3\}$; if an edge of snake q is input i_j or output o_j of some gate g of $\tilde{N}$, the color of snake q is j.

A snake whose input edge is assigned the value 0 by s_{in} may be deleted from the network, just as we did for X-networks. Next, assign each edge a color $\in \{1, 2, 3\}$ as follows: if edge e is output o_j of some gate, set color(e) equal to j; if e is an input edge of the network and is input o_j to some gate, set color(e) equal to j. (Edges that are both input and output edges of the network may be assigned an arbitrary color.) There are two difficulties to resolve—the network may have cyclic snakes, and there may be color conflicts, i.e., an edge with color j may be input i_k to a gate, with $j \neq k$.

Notice that $Y(1, 1, a) = (\bar{a}, \bar{a}, 0)$ and $Y(1, \bar{a}, 1) = (a, 0, a)$. Connect these two gates together by making the second output of the first gate be the second input of the second gate. Color the resulting circuit in the obvious fashion. The resulting circuit serves to "change the color" of an edge from 3 to 1. (Just ignore all outputs except output o_1 of the second gate.) Simple variants of this circuit can be used to change from any color j to any color k. Inserting such circuits in every noninput edge of the Y-network resolves all color conflicts and makes each snake acyclic. Let the color of a snake be the common color of its edges. The resulting network is $\tilde{N}$; it is easy to verify that it has all the desired properties.

We can read off an instance of Three-party Stable Marriage from network $\tilde{N}$, as follows. Introduce a participant for each snake of $\tilde{N}$, and read off the preference ordering of a participant from the order in which his snake participates in triples. A participant is a man, woman, or dog if and only if his snake is colored 1, 2, or 3 respectively. There are two difficulties: in the resulting instance, the numbers of the three different kinds of participants may not be equal, and each participant may not list all the triples he is supposed to. These difficulties may be resolved as in the proof of Lemma 7.2 by introducing extra snakes and appending a circuit. □

THEOREM 8.3. *Three-party Stable Marriage is NP-complete.*

Proof. By Lemma 8.2, it suffices to show that Y-NS is NP-complete. The Y-gate can simulate and be simulated by $\{NAND, COPY\}$, so Y-NS is equivalent to $\{NAND, COPY\}$-NS, which is known to be NP-complete [21]. □

Remark. Knuth [20] also proposes another formulation of Three-party Stable Marriage. In this "circular" version, men list women in order of preference, women list dogs, and dogs list men. A matching M of men, women, and dogs is stable if there is no destabilizing triple (m, w, d) such that man m prefers woman w to the woman he has been assigned to in M, woman w prefers dog d to the dog she has been assigned to in M, and dog d prefers man m to the man it has been assigned to in M. It is not known whether this problem has a polynomial-time solution.

9. Open problems. We seek new applications of the network approach to Stable Matching. We would like to find a fast parallel algorithm for Stable Matching. We would also like to find a fast parallel reduction from the Stable Marriage problem to the Assignment problem that preserves the structure of solutions.

Acknowledgments. This research has benefited from the ideas and suggestions of many. Ernst Mayr introduced me to the X-gate, and Danny Soroker introduced me to Stable Marriage. Discussions with Richard Anderson, Richard Cleve, Shaibal Roy, and Danny Soroker were very helpful in the early stages of this research. Marianne Baudinet helped me read portions of [20] and pointed me to Tarski's theorem; David Wolfe suggested that I look at Three-party Stable Marriage; and Alan Hoffman pointed out the importance of the nonalgorithmic proof of §6.2.1. Tomás Feder deserves very special thanks for simplifying the proof of Lemma 4.1, for many absorbing discussions, and for extending the network approach in wonderful ways in [5].

I would especially like to thank Ernst Mayr and Christos Papadimitriou for many useful discussions and for steadfast guidance and encouragement throughout the tenure of this research. I would also like to thank Andrew Goldberg, Dan Gusfield, Don Knuth, and Jeff Ullman for their comments at various stages of this research. This research has also been enriched by many intangible contributions from my fellow students and friends. Finally, thanks to the referees for their careful reading and constructive criticism.

REFERENCES

[1] G. Birkhoff, *Lattice theory*, Amer. Math. Soc. Colloq. Publ., 25, Amer. Math. Soc., Providence, RI, 1967.

[2] S. A. Cook, indirect personal communication.

[3] ———, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[4] S. A. Cook and M. Luby, *A simple parallel algorithm for finding a satisfying truth assignment to a* 2-CNF *formula*, Inform. Process. Lett., 27 (1988), pp. 141–145.

[5] T. Feder, *A new fixed point approach for stable networks and stable marriages*, J. Comput. System Sci., 45 (1992), pp. 233–284.

[6] ———, personal communication.

[7] D. Gale and L. S. Shapley, *College admissions and the stability of marriage*, Amer. Math. Monthly, 69 (1962), pp. 9–15.

[8] D. Gale and M. Sotomayor, *Some remarks on the stable matching problem*, Discrete Appl. Math., 11 (1985), pp. 223–232.

[9] L. M. Goldschlager, *The monotone and planar circuit value problems are logspace complete for P*, SIGACT News, 9 (1977), pp. 25–29.

[10] D. Gusfield, *Three fast algorithms for four problems in stable marriage*, SIAM J. Comput., 16 (1987), pp. 111–128.

[11] ———, *The structure of the stable roommate problem: efficient representation and enumeration of all stable assignments*, SIAM J. Comput., 17 (1988), pp. 742–769.

[12] D. Gusfield and R. W. Irving, *The Stable Marriage Problem: Structure and Algorithms*, MIT Press Ser. Found. Comput., MIT Press, Cambridge, MA, 1989.

[13] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[14] R. W. Irving, *An efficient algorithm for the stable roommates problem*, J. Algorithms, 6 (1985), pp. 577–595.

[15] ———, *On the Stable Room-Mates Problem*, Tech. Report CSC/86/R5, University of Glasgow, 1986.

[16] R. W. Irving and P. Leather, *The complexity of counting stable marriages*, SIAM J. Comput., 15 (1986), pp. 655–667.

[17] R. W. Irving, P. Leather, and D. Gusfield, *An efficient algorithm for the optimal stable marriage*, J. Assoc. Comput. Mach., 34 (1987), pp. 532–543.

[18] N. D. Jones, *Space-bounded reducibility among combinatorial problems*, J. Comput. System Sci., 11 (1975), pp. 68–85.

[19] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science, Vol. A, MIT Press/Elsevier, Cambridge, MA/Amsterdam, 1990.

[20] D. E. Knuth, *Mariages stables et leurs relations avec d'autres problèmes combinatoires*, Les Presses de l'Université de Montréal, Montréal, 1976.

[21] E. W. Mayr and A. Subramanian, *The complexity of circuit value and network stability*, Proc. 4th Annual Conference on Structure in Complexity Theory, 1989, pp. 114–123.

[22] C. Ng and D. S. Hirschberg, *Complexity of the stable marriage and stable roommate problems in three dimensions*, manuscript, 1988.

[23] G. Pólya, R. E. Tarjan, and D. R. Woods, *Notes on Introductory Combinatorics*, Birkhäuser, Basel, 1983.

[24] W. L. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[25] D. Soroker, personal communication.

[26] L. Stockmeyer, *Classifying the computational complexity of problems*, J. Symbolic Logic, 52 (1987), pp. 1–43.

[27] A. Subramanian, *The Computational Complexity of the Circuit Value and Network Stability Problems*, Ph.D. thesis, Department of Computer Science, Stanford University, 1990; Tech. Report STAN-CS-90-1311, Stanford University, 1990.

[28] A. Tarski, *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5 (1955), pp. 285–309.

[29] L. G. Valiant, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410–421.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 701–712, August 1994

WAIT-FREE CONSENSUS USING ASYNCHRONOUS HARDWARE*

BENNY CHOR[†], AMOS ISRAELI[‡], AND MING LI[§]

Abstract. This paper studies the wait-free consensus problem in the asynchronous shared memory model. In this model, processors communicate by shared registers that allow atomic read and write operations (but do not support atomic test-and-set). It is known that the wait-free consensus problem cannot be solved by *deterministic* protocols. A *randomized* solution is presented. This protocol is simple, constructive, tolerates up to $n - 1$ processors crashes (where n is the number of processors), and its *expected* run-time is $O(n^2)$.

Key words. asynchronous distributed systems, wait-free protocols, fault tolerance, randomized algorithms, consensus

AMS subject classifications. 68Q22, 90D10

1. Introduction. The problem of reaching consensus among different processors in a distributed environment [20] is one of the most fundamental problems whenever any type of cooperation is to be achieved. The nature of solutions to this problem depends on the properties of communication media, on the reliability of participating processors, and on their relative speeds. In this paper we investigate the consensus problem in a totally asynchronous system, where communication is carried out by shared registers that are atomic with respect to *read* and *write* operations, and up to $n - 1$ out of n processors may fail-stop (i.e., crash).

The consensus problem we study is the following multivalued problem: Every processor starts the protocol with an arbitrary input value (for example, an externally supplied variable or an internally computed constant). Upon termination, each processor decides on an output value. We have two requirements from the output. The first is that all processors that have terminated hold the same output value. The second is that the output value must be one of the input values of the processors. The consensus problem has been extensively studied in the asynchronous message passing model (e.g., [6], [24], [10]). The original version of this work [9] is the first one that studies and solves consensus in this asynchronous shared memory model.

It is convenient to think about all the *read* and *write* operations in terms of a global time model. In this model each such I/O operation takes place in a closed interval on the global-time axis. Atomicity of a register means that every set of *read*s and *write*s from/to this register is equivalent to a sequence in which each interval is shrunk to a distinct point, hence all these operations are totally ordered. We refer the reader to the works of Lamport [19], Herlihy and Wing [16], and Ben-David [5] for precise definitions of atomicity and linearizability. In particular, the techniques of [5] imply that when analyzing protocols that use atomic shared registers, the global time model can be used with no loss of generality.

*Received by the editors December 10, 1990; accepted for publication (in revised form) March 22, 1993. A preliminary version of this paper appeared in *Proceedings of the 6th Annual Association for Computing Machinery Conference on Principles of Distributed Computing*, August 1987, pp. 86–97.

[†] Department of Computer Science, Technion, Haifa 32000, Israel (benny@cs.technion.ac.il). Part of this author's research was carried out at Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, and was supported in part by National Science Foundation grant MCS81-21431. Research at the Technion was supported in part by United States–Israel Binational Science Foundation grant 88-00282, and by Technion Vice President for Research Fund–E. & J. Bishop Research Fund.

[‡] Department of Electrical Engineering, Technion, Haifa 32000, Israel. Part of this author's research was carried out at Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, and was supported in part by a Weizmann postdoctoral fellowship and National Science Foundation grant DCR-86-00379.

[§] Department of Computer Science, University of Waterloo, Ontario N2L 3G1, Canada. This author's research was supported in part by Office of Naval Research grant N00014-85-k-0445 at Harvard University, Cambridge, Massachusetts, and by National Science Foundation grant DCR-86-06366 at OSU.

We look for solutions to the consensus problem that satisfy the *wait-free termination* requirement. Wait-free termination means that every processor that is activated a sufficient number of times will decide and terminate. We would like to have a solution that guarantees that every schedule in which a processor is activated at least $k(n)$ times (for some $k(n)$ which is a function of n, the number of processors, but does not depend on the scheduler) leads to termination of that processor. This implies, in particular, that no processor needs to wait for other processors to take steps—it should terminate regardless of whether or not other processors were active in between its own steps (the output value could, however, depend on other processors' activity). Such a requirement is in accordance with the complete asynchrony of the system: It does not make sense to force the very fast processors to wait until a very slow processor makes a move. Furthermore, wait-free termination implies *resilience* to any number of processor crashes.

It is known that wait-free consensus cannot be achieved by *deterministic* protocols, even for systems with $n = 2$ processors. This impossibility result has been proven in the original version of this paper [9] and independently by Loui and Abu Amara [22]. It is also implicit in the work of Dolev, Dwork, and Stockmeyer [13]. All those proofs follow the ideas in the impossibility proof for the message passing model of Fischer, Lynch, and Paterson [17]. The gist of the proof is as follows: First, one shows that there are *bivalent* initial configurations of the system, namely, configurations that can lead to more than one decision value (under different schedulers). After establishing this fact, it is shown that starting from any bivalent configuration, there is an *infinite* scheduler that keeps the deterministic system in a bivalent state.

It is by now a well-known fact in the area of distributed computing that certain problems that cannot be solved by deterministic protocols do admit randomized solutions [24], [21], [6]. It is then only natural that in order to overcome the above-mentioned impossibility result, we employ a randomized protocol, allowing processors to toss coins. We present an efficient randomized protocol, that achieves consensus for systems of size n, using atomic single-writer multireader registers. The protocol is fairly simple and constructive, and its *expected* run-time is $O(n^2)$. This means that for any adversary scheduler, the system reaches a decision after $O(n^2)$ expected number of steps by *all* processors. The protocol uses unbounded size registers (though large values are actually written only with very low probability). The main usage of the unboundedness is to maintain a global order among processors. Processors who maintain larger values get preference over processors holding lower values. Coin flips are used to break possible ties among processors holding equal values.

We briefly discuss other approaches and developments. Loui and Abu-Amara [22] overcome the impossibility of deterministic consensus by using a much stronger communication primitive, namely, atomic test-and-set. Following the publication of the original version of our work [9], various improvements were made: One was designing protocols that operate in the presence of a stronger adversary model than the one used here. Another direction was the development of the so-called "bounded time stamps" [18], [14], and using them in consensus protocols with registers of bounded size. See §§2 and 4 for further details.

The remainder of this paper is organized as follows: In §2 we formally define the model, the class of admissible schedules, and the consensus problem. In §3 we present the protocol, and §4 contains some concluding remarks.

2. Model and definitions. In this section we define our model of asynchronous concurrent computation, the consensus problem, and the class of schedulers in which we are interested.

An asynchronous concurrent system is a collection of n processors. Every processor P is a (not necessarily finite) state automaton with an internal input register i_P and an internal output register o_P. The input register contains any value v taken from a set V, while the output register has initially the value $\perp$ ($\perp \notin V$) and could be changed once to a value in V. The set of all states of processor P will be denoted by S_P. The set S_P contains a set of states I_P that are the *initial states* of the processor P. States in S_P where o_P contains a value $\neq \perp$ are called the *decision states* of processor P. The set S_P might be infinite. In particular, this enables every internal state to include a description of the whole history of the computation of the processor P.

Processors communicate via *shared registers.* We use atomic single-writer multireader registers: Every shared register can be written by one processor and read by all other processors. Processors execute their programs by taking steps. A step consists of an internal operation, possibly involving coin tosses, and an input/output operation. In the model we consider, these two parts are executed as a single atomic step whenever the processor is scheduled.[1] Formally, every processor P takes steps according to its *transition function* T_P. Each step consists of a single input/output operation, followed by a state transition. The input/output operation could either be "read register r" or "write the value v to register r." In case the communication action is a read, the new state of P depends not only on the old state but also on the value read by this action. The transition function T_P could be either deterministic or randomized. In the latter case, for every state $s \in S_P$, there is a probability measure assigned to the next step. The choice of the actual step is done, according to these probabilities, only when the processor makes its next step. Given an asynchronous system as specified above, a *protocol* is a collection of n transition functions $T_1, \ldots, T_n$, one per processor.

A *configuration* C of the system consists of the state of each processor together with the contents of the shared registers. In an *initial configuration*, every processor is in an initial state, and all shared registers and output registers contain the default value $\perp$. The set of all configurations will be denoted by $\mathcal{C}$. A *step* takes one configuration to another by activating a single processor P. A *run* of length ℓ is a sequence of ℓ steps. Each run has an associated *schedule* that is a sequence of ℓ processors, numbered according to the order of processors that take steps in that run. We denote schedules, finite or infinite, by a list of processor numbers, e.g., $(2, 3, 3, 2, 1)$. If S is a finite schedule, then we denote by $S \circ i$, where i is any processor number, the schedule obtained from the schedule S by concatenating the number i to the end of S. We say that processor P is activated k times in a run if P appears k times in its schedule. The *history* $\mathcal{H}$ of a run is the sequence obtained by interleaving the sequence of configurations with the steps in the run, starting with the initial configuration. For a finite run, we refer to the last configuration in its history as the *current* configuration.

When arguing about randomized protocols, the power of the scheduler crucially depends on its adaptivity (see [8] for a discussion of this issue). *Adaptive schedulers* can use information derived from the state of the system and its history in making scheduling decisions. Formally, an admissible *scheduler* $\mathcal{S}$ in our system is a mapping from $\mathcal{H}$ into the set of n processors. Given the configuration of the system, the scheduler picks the next processor that is to take a step. The scheduler could either be a deterministic mapping or a randomized one. The scheduler is best viewed as an adversary that tries to prevent us from reaching our goal. Under the definition, this adversary scheduler is adaptive, and it has complete knowledge on the state of every processor and on the contents of the shared registers during the entire history.[2] In case the processors are randomized, the scheduler could also base its choices on the outcome

[1] An alternative approach separates the atomic operations to internal operations, input operations, and output operations.

[2] In the more refined level of atomicity, where internal operations, input operations, and output operations are separate, the adversary is even stronger. For example, it knows what a processor is about to write before scheduling that processor. For more details, see §4.

of past coin flips. We do not allow it, though, to be able to predict *future* randomized moves of the processors. This is a necessary requirement if randomization is to be helpful at all, and it is used in all algorithms where randomization is employed, e.g., [24], [21], [6]. In particular, in randomized protocols, a processor might be in a state in which the adversary does not know which input/output operation will be taken by that processor, before the action takes place. Given a history $\mathcal{H}$ and a scheduler $\mathcal{S}$, the runs that can be produced by $\mathcal{S}$, extending $\mathcal{H}$, on some possible randomized choices are called the runs *compatible* with $\mathcal{H}$ and $\mathcal{S}$. Notice that if both processors and scheduler are deterministic, then there is a single compatible run extending $\mathcal{H}$.

We say that a configuration C is reachable from history $\mathcal{H}$ with schedule $\mathcal{S}$ if there is a run compatible with $\mathcal{H}$ and $\mathcal{S}$ that leads to configuration C. We say that a configuration has a decision value v if some processor P is in a decision state with its output register o_P containing $v \neq \perp$.

A *randomized consensus protocol* is designed for an asynchronous system of n processors ($n \geq 2$). The protocol specifies a set V of possible inputs whose cardinality is at least two (otherwise the problem is trivial). It is required to satisfy the following properties:

(1) **Consistency:** for every schedule, no configuration reachable from an initial configuration has more than one decision value.

(2) **Nontriviality:** if processor P has decided on value v in a run, then v is an input value for at least one processor.

(3) **Randomized wait-free termination:** each processor must decide after taking a finite expected number of steps. Formally, there is a probability function f from the natural numbers into the interval [0, 1] $\left(\sum_{k=1}^{\infty} f(k) = 1\right)$, satisfying $\sum_{k=1}^{\infty} kf(k) < \infty$, such that for every initial configuration C_0 and for every admissible scheduler, if a processor P was activated k times by the scheduler, then the conditional probability that P is in a decision state, conditioned on P not being in a decision state after its previous activation, is at least $f(k)$.

We required that randomized consensus protocols will never err. The randomization effects only the running time of the protocol and not its correctness. There could be a positive probability for arbitrary long nonterminating runs, but this probability should be very small (converging to 0 with the length of the run), so that the expected running time is bounded.

3. Wait-free consensus protocol. The high-level structure of the protocol is as follows: In every point of an execution, each processor holds a preferred value (which is a potential decision value) and a confidence level (which is a nonnegative integer). Initially, the confidence level is 0, and the preferred value of the processor is its input value. Both the preferred value and the confidence level are written by each processor into a shared register, which can be read by all others. Processors compare their confidence levels, and if a large enough gap forms, the leading processor decides on its preferred value. In case of ties, processors increment their confidence level. In order to prevent live locks, where competing processors concurrently increment their confidence ad infinitum, coin flips are used. Confidence levels can possibly reach any nonnegative integer, which means that registers of unbounded size are used by the protocol. There is a positive (though very small) probability for very large numbers to be written into the shared registers. This probability decreases to 0 when the numbers increase to infinity.

To simplify the description of the protocol, we will say that a processor is on node i if its confidence level is i. The initial node is node 0. Before deciding and terminating, the processor moves to a special node, denoted by ∞ (this move facilitates the design and analysis of the protocol). Each processor starts execution by writing its input value in its register while staying on the start node. The steps of each processor in any given history H of the protocol are divided into *phases*. In each *phase* a processor reads the registers of all other processors, computes a new value, and writes it in its own register. A processor decides if it is at least two nodes ahead of all other processors with contending values. Thus, by the time of decision, processors with contending values are at least one step behind and will change their preferred value to that of the leading processor. There could be a situation with ties. The protocol resolves ties by having the option of advancing (to the next node) or not advancing, according to the outcome of coin tossing. This is where the use of randomization overcomes the deterministic impossibility result. In a bivalent configuration, only some of the choices made by some processors lead to another bivalent configuration. Other choices could lead to a univalent configuration. The adversary does not know which choices the processor will make *before* scheduling it, because the choice is made by flipping a coin.

If the coin used by every processor (in choosing whether to advance or not) would be unbiased, then with high probability, about half the contenders would advance. Those lagging behind would then join them, and again we would be in a tied situation. While such protocol, using unbiased coins, would satisfy the requirements of randomized consensus, it would lead to exponential (in n) expected running time (for an appropriate adversary strategy). To be more efficient, our protocol tries to have, with high probability, only one successful advancement out of n attempts. Leading processors in a tied situation flip a *biased* coin and advance only with small probability. Intuitively, this probability should be $\theta\left(\frac{1}{n}\right)$. The specific value we use, $\frac{1}{2n}$, is based on calculations done to minimize the expected running time.

It is dangerous to let a lagging processor decide, even if all leading processors have the same preferred value. The reason is that another processor might advance substantially after its value was last read by the lagging processor. The lagging processor, who thinks all leading processors have the same value, would in fact be wrong. Therefore, in our protocol, processors lagging behind never decide, and they always advance. A lagging processor who sees all leading processors with the same preferred value changes its own value. If the lagging processor sees conflicts at the top, it retains its old preferred value. In both cases, the lagging processor advances. If it is no more than two nodes behind the maximum, it advances by one. If it more than two nodes behind the maximum, it "jumps" to a point that is the maximum minus two. This shortcut allows the protocol to converge quickly to a decision, even if it starts from a configuration where one processor is way behind other active processors (e.g., if it just woke up). Thus, our upper bound on the expected running time will be valid starting from *every* reachable configurations, and not just the initial one.

To continue the description of the protocol, some definitions are introduced. Assume that processor P_i has just finished all the read steps in its jth phase, a phase we denote by ϕ_j^i. Let maxnode_j^i be the maximum node on which P_i sees a processor during ϕ_j^i (including itself). Using the data collected on ϕ_j^i, P_i computes two sets of processors, L and AL. The set L (the leaders) contains the processors whose node, as read by P_i , is maxnode_j^i. The set AL (the almost leaders) is the set of processors whose node, as read by P_i , is $\text{maxnode}_j^i - 1$. Denote by L_j^i (AL_j^i) the set L (AL) computed by P_i in phase ϕ_j^i.

A processor P_i terminates after the write step of phase ϕ_j^i in one of two cases:

T_1 If another processor has already terminated. The decision value is the value of the terminating processor.

T_2 If P_i itself is in the set L_j^i and all processors in the set $L_j^i \cup AL_j^i$ have the same *pref*. The decision value is the common *pref*.

We say that a processor P_i is *committed to terminate* in phase ϕ_j^i if it completed all the read steps of the phase and one of the termination conditions T_1 or T_2 holds (so its next step is a write, after which P_i decides and terminates). If none of the termination conditions holds, then P_i either moves to a new node or it stays put. The motivation behind the protocol design is to create a single leader. If P_i is a leader (that is, $P_i \in L_j^i$), then the new node to which

```
        $newreg := (input, 0)$
        $reg_1$ :=write($newreg$)
        repeat
              $v_1 := newreg$
              for $i := 2$ through $n$ do $v_i$ :=read ($reg_i$) od
              $maxnode := \max_{1 \le i \le n}\{v_i.node\}$
              $L := \{P_i : v_i.node = maxnode\}$
              $AL := \{P_i : v_i.node = maxnode - 1\}$
($T_1$)       if $\exists i \; v_i.node = \infty$
                then $reg_1$ := write($v_i.pref$, $\infty$), decide $v_i.pref$ and halt.
($T_2$)       elseif $P_1 \in L$ and $pref$ of all processors in $L \cup AL = v_1.pref$
                    then $reg_1$ :=write($v_1.pref$, $\infty$), decide $v_1.pref$ and halt.
              elseif $P_1 \in L$
                    then
                          $newreg.pref := v_1.pref$
                          $newreg.node := v_1.node + 1$
                          toss a biased coin with $1/2n$ probability of $heads$
                          if $tails$ (this occurs with probability $1 - 1/2n$)
                            then $newreg := v_1$ (retain old value)
                          endif
              elseif $maxnode - v_1.node \le 2$
                    then
                          $newreg.node := v_1.node + 1$
                          if all leading processors have the same $pref$
                            then
                                $newreg.pref :=$ $pref$ of leading processor
                            else
                                $newreg.pref := v_1.pref$
              else ($maxnode - v_1.node \ge 3$)
                    $newreg.node := maxnode - 2$
                    $newreg.pref :=$ $pref$ of processor with minimum index in $L$
              endif
              $reg_1$ :=write($newreg$)
        until decision is made
```

FIG. 1. *The n processor protocol (for P_1).*

it can move is the successor of its node. However, it moves to its successor node only with probability[3] $1/2n$. With probability $1 - 1/2n$, P_i stays on its current node. In both cases, the leader P_i retains its old preferred value.

If P_i is not a leader, then it always moves to a new node. Let k denote P_i's node during ϕ^i_j. If $\text{maxnode}^i_j - k \le 2$, then P_i moves to the successor of its node, $k + 1$. In this case, P_i's new preferred value is determined as follows: If all processors in L^i_j have the same *pref*, this value is the new *pref* of P_i. If this is not the case, then P_i keeps its own *pref*.

If P_i is not a leader and $\text{maxnode}^i_j - k \ge 3$, then P_i moves to node $\text{maxnode}^i_j - 2$. In such case, we say that P_i *jumps*. In case of a jump, the new *pref* of P_i is the *pref* of the processor in L^i_j with the minimal index.

LEMMA 3.1. *Let H_0 be an initial segment of a history H of an arbitrary execution of the protocol. If in H_0 no processor reaches node i with preferred value v, then for any node $j > i$, no processor reaches j, in H_0, with preferred value v.*

Proof. Assume, toward a contradiction, that H is the history of an execution not satisfying the lemma. Let H_0 be an initial segment of H of minimal length that violates the lemma. This

[3] This specific probability $1/2n$ was chosen in order to optimize the expected running time.

means that there is a node i such that in H_0 no processor reaches i with preferred value v and there is a processor P_k that reaches a node $j > i$ with preferred value v.

Let ϕ_ℓ^k be the phase in which P_k moves to node j. If P_k jumps to j, then, by the protocol, at least one processor $P_m \in L_\ell^k$ has been on node $j + 2$ with preferred value v before this jump. Since $j+2 > i$, this contradicts the minimality of H_0. We can therefore assume that P_k moves to node j from node $j - 1$ without jumping. First, we show that the preferred value of P_k at $j - 1$ is $v' \neq v$. If $j - 1 = i$ then, this is true by our assumption that in H_0 no processor prefers the value v on node i. If $j - 1 > i$, then this is true because of the minimality of H_0. Having established this claim, we observe that by the protocol, in order for P_k to change its preferred value from v' to v while moving from $j-1$ to j it has to see all processors in L_ℓ^k with preferred value v. We now show that for any possible value of maxnode_ℓ^k this is impossible. In ϕ_ℓ^k P_k is on $j - 1 \geq i$. Therefore $\text{maxnode}_\ell^k \geq j - 1 \geq i$. If $\text{maxnode}_\ell^k = i$, then by our assumption no processor has preferred value v on i. If $\text{maxnode}_\ell^k > i$, then as we have shown above, no processor in L_ℓ^k has preferred value v. □

LEMMA 3.2. *Let P_k be the first processor reaching node i with preferred value v. Then P_k does not jump to i, and its preferred value on node $i - 1$ is also v.*

Proof. Let ϕ_ℓ^k denote the phase in which P_k moves to node i, and let H_0 be the initial segment of the history of the execution that ends with the last read step of P_k in ϕ_ℓ^k. In H_0 no processor has reached node i with preferred value v. Thus, by Lemma 3.1, no processor has reached any node $j \geq i$ in H_0 with preferred value v. Therefore P_k sees no leader on any node $j \geq i$ with preferred value v during ϕ_ℓ^k. According to the protocol, P_k does not jump to node i with preferred value v in ϕ_ℓ^k. Since P_k reaches node i with preferred value v, this argument implies that P_k does not jump to node i.

Assume, by way of contradiction, that P_k changes its preferred value from v' to v while moving from $i - 1$ to i at the end of ϕ_ℓ^k. This happens only if *pref* of all processors in $L_{k,\ell}$ is v. Consider the following cases:

Case 1: $\text{maxnode}_\ell^k = i - 1$. In this case $P_k \in L_\ell^k$. By the protocol P_k keeps its preferred value while moving to i. Contradiction.

Case 2: $\text{maxnode}_\ell^k = i$. In this case the preferred value of all processors in L_ℓ^k does not equal v, since we assumed that P_k is the first processor reaching i with preferred value v. Contradiction.

Case 3: $\text{maxnode}_\ell^k > i$. In H_0 no processor has been on node i with preferred value v, and by Lemma 3.1, in this execution no processor has preferred value v on maxnode_ℓ^k before the completion of ϕ_ℓ^k. Contradiction.

We conclude that P_k prefers the value v during ϕ_ℓ^k, its last phase on node $i - 1$. By the protocol, processors retain their preferred values when staying on the same node. This means that P_k prefers the value v during all phases it executes while residing on node $i - 1$. □

LEMMA 3.3. *Let H be the history of an arbitrary execution of the protocol. Let P_j be a processor committed to terminate, in H, by T_2. If P_j is committed to terminate on i with decision value v, then in H, no processor reaches i with a preferred value $v' \neq v$.*

Proof. Assume, by way of contradiction, that P_ℓ is the first processor reaching i with preferred value $v' \neq v$. By Lemma 3.2, P_ℓ moves to node i without jumping, and the preferred value of P_ℓ on $i - 1$ is also v'. Let ϕ_k^j be the phase in which P_j is committed to terminate with v. At the beginning of ϕ_k^j, P_ℓ is on node $m < i - 1$. (Otherwise P_j sees P_ℓ as either a leader or an almost leader during phase ϕ_k^j, and condition T_2 does not hold.) After P_ℓ advances to $i - 1$ it starts a new phase ϕ_r^ℓ. In ϕ_r^ℓ, P_ℓ reads the values of all processors. The leaders in L_r^ℓ are on a node $\geq i$. The value preferred by all these leaders before P_ℓ moves there is v, as the only value preferred at i is v. By the protocol P_ℓ takes v as its new preferred value and advances to i. Contradiction. □

LEMMA 3.4. *Let H be the history of an arbitrary execution of the protocol. Assume that in H a single processor P_i moves from node.m to node $m + 1$ as a result of a successful coin toss. (That is, all other processors that tried to move from m to $m + 1$ as a result of a coin toss fail to do so.) If P_i's preferred value on node m is v, then all processors reaching node $m + 1$ in H have v as their preferred value on node $m + 1$.*

Proof. Assume, by way of contradiction, that the lemma does not hold. Let P_j $(i \neq j)$ be the first processor reaching node $m + 1$ with preferred value v' $(v' \neq v)$ in H. By Lemma 3.2, P_j does not jump to node $m + 1$. By the supposition, P_j does not flip a coin when moving from node m to node $m + 1$. This implies, by the protocol, that P_j is not one of the leaders during the phase ϕ_ℓ^j in which it moves to node $m + 1$ (i.e., $P_j \notin L_\ell^j$). Since all processors who move to node $m + 1$ in H *before* P_j's move prefer the value v on node $m + 1$, it follows from Lemma 3.1 that all leading processors (those in L_ℓ^j) prefer the value v. All these leaders are on a node $\geq m + 1$ and according to the protocol P_j moves to $m + 1$ with preferred value v in phase ϕ_ℓ^j—contradiction. □

LEMMA 3.5. *Let H be the history of an arbitrary execution of the protocol. Assume that in H, processor P_i is the first processor who moves from node m to node $m + 1$ as a result of a successful coin toss. Then after this write step of P_i, any other processor P_j makes at most one attempt to move from node m to node $m + 1$ as a result of a coin toss.*

Proof. Consider the execution after P_i's move. If P_j succeeds in its first attempt to move from node m to node $m + 1$ as a result of a coin toss, then we are done. If P_j fails, then it stays on node m. In the phase following the failed attempt, P_j is not a leader (as it sees P_i at least one node ahead). By the protocol, P_j does not flip a coin in its next phase on node m and moves to a new node in this phase. □

THEOREM 3.6. *The n processor protocol is consistent.*

Proof. Let H be an arbitrary history of the protocol. It is easy to see that a processor terminates by T_1 with value v only if some other processor terminated earlier with v by T_2. Therefore, it suffices to show that in H all processors that terminate by T_2 have the same decision value. Let i be a minimal node on which some processor is committed to terminate by T_2. Without loss of generality assume that P_j terminates on i with value v. In order to prove that the protocol is consistent we will show that no other processor is committed to terminate on any node with a value $v' \neq v$. By the protocol, the first processor committed to terminate with value v' is committed by T_2. Since i is a minimal node on which any processor is committed to terminates by T_2, no processor is committed to terminate by T_2 on any node $k < i$. By Lemma 3.3 no processor reaches i with any preferred value $v' \neq v$. By the protocol, this implies that no processor is committed to terminate by T_2 on node i with decision value v'. This also implies, by Lemma 3.1, that no processor reaches any node $k > i$ with preferred value $v' \neq v$. Therefore, by the protocol, no processor is committed to terminate by T_2 with decision value $v' \neq v$. □

We now proceed to analyze the expected running time of the multiprocessor protocol.

THEOREM 3.7. *Let C be any reachable configuration of the n processor system and $\mathcal{A}$ an arbitrary adversary scheduler. If $\mathcal{A}$ schedules the processors such that at least $15n$ entire phases are executed following C, then with probability ≥ 0.4534, at least one processor decides and terminates.*

Proof. Let m $(m \geq 0)$ be the maximal node on which any processor resides at C. By the protocol, any processor P_i that executes an entire phase following C finds that some processor resides on a node $\geq m$ and will subsequently move to a node $j \geq m - 2$ in the write step of this phase. After completing two additional phases, P_i reaches node $j \geq m$. (Recall that if a processor is not among the leaders in some phase, then it does not flip a coin and traverses at least one edge in the **write** step of that phase.)

Thus, of the $15n$ entire phases that are executed following C, at most $3n$ are executed by processors residing on nodes smaller than m. Therefore, at least $12n$ entire phases are executed, following C, by processors residing on nodes greater than or equal to m.

Consider the first processor P_i that is scheduled to make a **write** step while residing on node m following C. If some other processor has already decided before this write step of P_i, then we are done. If no other processor has yet decided, then since m is the maximal node in C, the value maxnode that P_i maintains at the time of this write equals m. If P_i decides and moves to ∞, then we are done. Otherwise, according to the protocol, P_i flips a coin when making its write step, and if it succeeds (this happens with probability $1/2n$), it moves to node $m + 1$.

We base our analysis on the following two events:

- E_1: Of the first $4n$ attempts to move from node m to node $m + 1$ as a result of a coin toss following C, *exactly* one succeeds. Subsequent to the successful move, all attempts to move from node m to node $m + 1$ as a result of a coin toss fail.
- E_2: Of the first $4n$ (or fewer) attempts to move from node $m + 1$ to node $m + 2$ as a result of a coin toss following C, *at least* one succeeds.

Using Lemma 3.5, the number of subsequent attempts to toss a coin on node m, after the first successful toss on node m, is less than or equal to $n - 1$. Since the adversary is unable to predict the outcome of a write that uses coin tossing before the action takes place, we have

$$Pr(E_1) \geq \left(\overbrace{\frac{1}{2n}}^{\text{success in 1st}} + \overbrace{\left(1 - \frac{1}{2n}\right)\frac{1}{2n}}^{\text{success in 2nd}} + \dots + \overbrace{\left(1 - \frac{1}{2n}\right)^{4n-1}\frac{1}{2n}}^{\text{success in }4n\text{th}} \right)$$
$$\cdot \overbrace{\left(1 - \frac{1}{2n}\right)^{n-1}}^{\text{no subsequent success}}$$
$$= \left(1 - \left(1 - \frac{1}{2n}\right)^{4n}\right)\left(1 - \frac{1}{2n}\right)^{n-1}.$$

If E_1 occurs, then at most $4n + n - 1 = 5n - 1$ of the entire phases that are executed on nodes $\geq m$ following C involve an attempt to move from node m to $m + 1$ by tossing a coin. At most $n - 1$ additional phases can involve moving from node m to $m + 1$ without tossing a coin. Thus overall, if E_1 occurs, then at most $6n - 2$ of the entire phases that are executed on nodes $\geq m$ following C are executed by processors residing on node m. This implies that at least $(12 - 6)n + 2 = 6n + 2$ entire phases are executed, following C, by processors residing on nodes greater than or equal to $m + 1$.

Consider the first processor P_i that is scheduled to make a **write** step while residing on node $m + 1$ following C. If some other processor has already decided before this write step of P_i, then we are done. If no other processor has yet decided, then since m is the maximal node in C, the value maxnode that P_i maintains at the time of this write must equal $m + 1$. If P_i decides and moves to ∞, then we are done. Otherwise, according to the protocol, P_i flips a coin when making its write step, and if it succeeds (this happens with probability $1/2n$), it moves to node $m + 2$. If P_i does not succeed, then by the same reasoning the next processor that resides on node $m + 1$ flips a coin when it is scheduled to write , and so on. Thus, until at least one processor succeeds, all processors that reside on node $m + 1$ try to move to node $m + 2$ by flipping a coin. The probability that out of the first $4n$ attempts at least one is successful satisfies

$$Pr(E_2 \mid E_1) \geq 1 - \left(1 - \frac{1}{2n}\right)^{4n} .$$

The sequence $\{(1 - \frac{1}{2n})^{n-1}\}_{n=2}^{\infty}$ is monotonically decreasing to the limit $\frac{1}{\sqrt{e}}$. The sequence $\{1 - (1 - \frac{1}{2n})^{4n}\}_{n=2}^{\infty}$ is monotonically decreasing to the limit $1 - \frac{1}{e^2}$ (the limits can easily be verified, using the fact that $\{(1 - \frac{1}{k})^k\}_{k=2}^{\infty}$ monotonically increases to $\frac{1}{e}$). Combining these properties with the two inequalities above, we have (for $n \geq 2$)

$$\begin{aligned} Pr(E_2 \cap E_1) &\geq \left(1 - \left(1 - \frac{1}{2n}\right)^{4n}\right)^2 \left(1 - \frac{1}{2n}\right)^{n-1} \\ &\geq \left(1 - \frac{1}{e^2}\right)^2 \cdot \frac{1}{\sqrt{e}} \\ &> 0.4534 . \end{aligned}$$

If E_2 occurs, then at least one of the first $4n$ (or fewer) attempts succeeds. Using Lemma 3.5, the number of subsequent attempts to toss a coin on node $m + 1$ after the first successful toss is less than or equal to $n - 1$. At most $n - 1$ additional phases can involve moving from node $m + 1$ to $m + 2$ without tossing a coin. Thus overall, at most $6n - 2$ of the entire phases that are executed on nodes $\geq m + 1$ following C are executed by processors residing on node $m + 1$. This implies that at least $(6n + 2) - (6n - 2) = 4$ entire phases are executed, following C, by processors residing on nodes greater than or equal to $m + 2$.

In particular, it follows that at least one processor P_i completes a phase, including a write step, while residing on node $m + 2$. By Lemma 3.4, if E_1 occurs, all processors reaching node $m + 1$ have the same preferred value v on node $m + 1$. But since on node $m + 1$ all processors prefer the same value v, Lemma 3.1 implies that on $m + 2$ all processors prefer v as well. Thus the first processor P_i that completes a phase while residing on node $m + 2$ finds out during that phase that all leaders (processors on $m + 2$) and almost leaders (processors on $m + 1$) prefer v. P_i thus moves to the decision node ∞ by T_2, and terminates.

Therefore, with probability at least 0.4534, at least one processor terminates after $15n$ phases are completed following C. □

Using Theorem 3.7, we can easily give an upper bound on the expected time until some processor decides, starting from any reachable configuration C. Dividing the execution into blocks such that in each block exactly $15n$ phases are completed, we know that the probability of termination in each block is at least 0.4534. The expected number of entire phases is thus $15n/0.4534 < 33.1n$ operations. In terms of *elementary* operations (atomic read, atomic write), each entire phase involves exactly n elementary operations. At most $n(n - 1)$ initial elementary operations can belong to phases whose execution have already begun. Thus, the expected number of elementary operations until at least one processor decides is at most $33.1n^2 + n(n - 1) < 35n^2$. This implies the following.

THEOREM 3.8. *The n processor protocol is a randomized wait-free consensus protocol. Starting from any reachable configuration, the expected number of elementary steps until at least one processor decides is less than $35n^2$.*

4. Concluding remarks.

The analysis of the expected running time of our n processors protocols relied on the inability of the adversary to predict the outcome of a write that uses coin tossing, before the action takes place. Following the publication of the original version of our paper [9], Abrahmson [1] considered a stronger adversary model, where the outcome of the coin toss that is used in the next write step is known to the adversary before the step takes place. In this adversary model, the scheduling choices can be based on the outcome of the coin. Abrahmson modified our protocol and produced one that works in the presence of the strong adversary but has exponential ($2^{O(n^2)}$) expected running time. Subsequently, this was dramatically improved by Aspnes and Herlihy [3], who designed an efficient wait-free consensus protocol for this strong adversary model, with $n^{O(1)}$ expected running time. The protocol of Aspnes and Herlihy employs the same basic structure of our protocol, namely an incremental walk on the line of nonnegative integers. It introduces novel ideas from the theory of random walks in the implementation of the coin flips. Improved algorithms that use *bounded* shared registers and work in the presence of the strong adversary were later designed by Attiya, Dolev, and Shavit [4], Aspnes [2], and Saks, Shavit, and Wohl [25]. (Most of these algorithm solve the somewhat simpler problem of *binary* consensus, where the input set is $\{0, 1\}$.) The expected running time of the later protocol is $\Theta(n^3)$ elementary steps. This has subsequently been improved by Bracha and Rachman [7] to an $O(n^2 \log n)$ consensus protocol. Despite these improvements, our protocol remains the most efficient of which we know for the model considered in this paper and is a strong candidate for practical consensus protocols. By bounding the size of the shared registers in our protocol to, say, 128 bits per processor, we get a protocol that still never errs and has probability less than 2^{-56} of nontermination.

We view the possibility of achieving wait-free consensus as a fundamental tool in shared memory systems and believe it is only the first step in a promising direction. The subsequent results of Herlihy [15] and Plotkin [23] on wait-free implementation of sequential objects and of Chor, Moscovici, and Nelson [11], [12] on solvability of distributed decision tasks and distributed interactive tasks seem to support this belief.

Acknowledgments. We thank Maurice Herlihy for pointing out an error in the original version of the protocol. Discussions with Nissim Francez and Lenny Pitt that helped initiating this work are also acknowledged.

REFERENCES

[1] K. Abrahmson, *On achieving consensus using a shared memory*, Proc. 7th ACM Conference on Principles of Distributed Computing, August 1988, pp. 291–302.

[2] J. Aspnes, *Time- and space-efficient randomized consensus*, Proc. 9th ACM Conference on Principles of Distributed Computing, August 1990, pp. 325–331.

[3] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[4] H. Attiya, D. Dolev, and N. Shavit, *Bounded polynomial randomized consensus*, Proc. 8th Annual ACM Symposium on Principles of Distributed Computing (1989), pp. 281–293.

[5] S. Ben-David, *The global time assumption and semantics for concurrent systems*, Proc. 7th Annual ACM Symposium on Principles of Distributed Computing (1988), pp. 223–231.

[6] M. Ben-Or, *Another advantage of free choice: completely asynchronous agreement protocols*, Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (1983), pp. 27–30.

[7] G. Bracha and O. Rachman, *Randomized consensus in expected $O(n^2 \log n)$ operations*, Proc. 5th Internat. Workshop on Distributed Algorithms (1991), Lecture Notes in Comput. Sci. 579, Springer-Verlag, New York, pp. 143–150.

[8] B. Chor and C. Dwork, *Randomized algorithms for distributed agreement—A survey*, Advances in Computing Research, Vol. 5, Randomness and Computations, S. Micali, ed., JAI Press, 1989, pp. 443–497.

[9] B. Chor, A. Israeli and M. Li, *On Processors Coordination Using Asynchronous Hardware*, Proc. 6th ACM Conference on Principles of Distributed Computing, 1987, pp. 86–97.

[10] B. Chor, M. Merritt, and D. Shmoys, *Simple constant-time consensus protocols in realistic failure models*, J. ACM, 36 (1989), pp. 591–614.

[11] B. Chor and L. Moscovici, *Solvability in asynchronous environments*, Proc. 30th IEEE Conference on the Foundations of Computer Science, 1989, pp. 422–427.

[12] B. Chor and L. Nelson, *Resiliency of interactive distributed tasks*, Proc. 10th ACM Conference on Principles of Distributed Computing, 1991, pp. 37–49.

[13] D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[14] D. Dolev and N. Shavit, *Bounded concurrent time stamps systems are constructible*, Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 454–466.

[15] M. Herlihy, *Impossibility and universality results for wait free synchronization*, Proc. 7th Annual ACM Conference on Principles of Distributed Computing, 1988, pp. 276–290.

[16] M. Herlihy and J. Wing, *Linearizability: A correctness condition for shared objects*, ACM Trans. Program. Languages and Systems, 12 (1990), pp. 463–492.

[17] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[18] A. Israeli and M. Li, *Bounded time stamps*, Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 371–382.

[19] L. Lamport, *On interprocess communication*, Distributed Computing, Vol. 1, 1986, pp. 77–101.

[20] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.

[21] D. Lehmann and M. Rabin, *On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem*, Proc. 8th Principles of Programming Languages 1981, pp. 133–138.

[22] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, Advances in Computing Research, JAI Press, 1987, pp. 163–183.

[23] S. Plotkin, *Sticky bits and the universality of consensus*, Proc. 8th Annual ACM Conference on Principles of Distributed Computing, August 1989, pp. 159–175.

[24] M. Rabin, *The choice coordination problem*, Acta Inform., 17 (1982), pp. 121–134.

[25] M. Saks, N. Shavit, and H. Wohl, *Optimal time randomized consensus — making resilient algorithms fast in practice*, Proc. 2nd ACM Symposium on Discrete Algorithms, 1991, pp. 351–362.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 713–737, August 1994

INFERRING EVOLUTIONARY HISTORY FROM DNA SEQUENCES*

SAMPATH K. KANNAN† AND TANDY J. WARNOW‡

Abstract. One of the longstanding problems in computational molecular biology is the *Character Compatibility Problem*, which is concerned with the construction of phylogenetic trees for species sets, where the species are defined by characters. The character compatibility problem is NP-Complete in general. In this paper an $O(n^2k)$ time algorithm is described for the case where the species are described by quaternary characters. This algorithm can be used to construct phylogenetic trees from DNA sequences.

Key words. algorithms, graphs, evolutionary trees, evolution

AMS subject classifications. 05C05, 05C85, 68Q25, 92-08, 92B05, 92D15, 92D20

1. Introduction. Given a set S of n species, a *phylogenetic tree*, or, more simply, a *phylogeny* or *evolutionary tree*, is a rooted tree with n leaves that describes the evolution of the species in S from a common ancestor. The internal nodes represent ancestral species, and the leaves represent the species in S.

One standard way to describe species is through the use of characters. A *character* is an equivalence relation on the set S of species, partitioning the species set into the distinct equivalence classes, which we will call the *character states*. Since the number of possible states of any character is finite over any finite species set, we can number the states of each character. If there are n species and k characters, we can represent our data by an $n \times k$ integer matrix M, where the entry M_{ij} is the state of species s_i for the jth character. The ith row in this matrix is a vector over Z^k (where Z denotes the integers) that is identified with the species s_i.

In any phylogeny T for the species set S each leaf has a vector of character states in Z^k associated with it. We also assign to each internal node of the tree T a vector in Z^k thereby hypothesizing the character states of the ancestral species in our phylogeny. The desired property for this phylogeny (and assignment of vectors to internal nodes) is that, for each state of each character, the set of nodes in the phylogeny having that state should form a subtree. That is, the subgraph induced by those nodes should be a connected graph. Such a phylogeny is called a *perfect phylogeny*. This is also expressed by saying the phylogeny is convex for each character, or simply that it has the *convexity* property.

Given a phylogeny T, Fitch [17] has shown that one can efficiently determine whether there is an assignment of character vectors to the internal nodes so as to make T perfect. Thus the crux of the problem is the choice of phylogeny.

When a perfect phylogeny exists for a set C of characters defining a species set S, we say the character set is *perfectly compatible*, or, simply, *compatible*, on the species set S. In the literature it is common to drop the reference to the species set S and simply say the character set C is *compatible*. When there is no possibility of confusion about the species set, we shall use this convention.

We can now state the following.

Perfect phylogeny problem. Given a set of species defined by a set of characters, determine whether a perfect phylogeny exists.

*Received by the editors September 24, 1991; accepted for publication (in revised form) April 27, 1993.

†Department of Computer Science, University of Arizona, Tucson, Arizona 85721-0001. This author's research was supported in part by National Science Foundation grants CCR-8513926 and CCR91-08969.

‡Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania 19104-6389. This author's research was supported by National Science Foundation grant DMS 90-05833 and Department of Energy grants DE-FG03-90ER60999 and DE-AC04-76DP00789. This work was performed in part at Sandia National Laboratories.

This problem is also referred to in the literature as the *Character Compatibility Problem.*

What we will describe is an $O(n^2k)$ divide-and-conquer algorithm that will determine whether a set of n species defined by k quaternary characters has a perfect phylogeny; when a perfect phylogeny exists, the algorithm will construct one. If each species in S is defined by a DNA sequence and the DNA sequences are all perfectly aligned (so that there are no gaps), then each position in the sequences represents a character. Thus we have the quaternary character compatibility problem, since each position must take on one of only four possible values (A, C, T, or G). We can apply the algorithm to this case and determine whether a perfect phylogeny exists for species defined by perfectly aligned DNA sequences.

Unfortunately, in practice the set of characters is seldom perfectly compatible. We can still use our algorithm to find *maximal* sets of compatible characters. This can be done by iteratively adding a compatible character to a set of mutually compatible characters until no further characters can be added. This procedure takes time $O(n^2k^2)$. By contrast, the problem of finding a *maximum* set of compatible quaternary characters is *NP-Hard* [7].

The structure of the remainder of the paper is as follows. We discuss the history of the character compatibility problem in §2. Essential concepts that are used in the algorithm for quaternary character compatibility are defined in §3. We show how we reduce to subproblems for the trinary character compatibility problem in §4. The reduction to subproblems for quaternary characters is described in §5. We conclude with a discussion of further problems in §6.

2. History of the character compatibility problem. The character compatibility approach to phylogenetic tree construction is related to parsimony and compatibility criteria. Parsimony criteria seek to find a tree of minimum size, where the size of the tree is the number of character state changes indicated by the edges. Compatibility criteria seek instead to keep as many characters as possible while eliminating others, so that with respect to the characters that are kept, it is possible to construct a perfect phylogeny. Each approach has its proponents, but essentially they agree in their philosophy, which is that *the tree of minimum evolution is the right tree.*

This approach to phylogenetic tree construction originated in the 1960s. (Edwards and Cavalli-Sforza are usually credited with introducing parsimony in their papers from the early 1960s [9], [10]); variations on the basic parsimony criterion were also proposed ([22],[23], [24], [4] and others).

The concept of characters being perfectly compatible with a given tree arose naturally from these approaches, in that it represents the best *possible* tree (i.e., having minimum possible evolution on the tree). This objective was championed by LeQuesne in a series of papers [23], [24], [25], [26], and later given its firm mathematical foundation in the literature by two papers of Estabrook and McMorris [11], [12].

Since construction of phylogenies using compatibility criteria was shown to be an *NP-hard* problem by Day and Sankoff in [7] and many variations of parsimony were also shown to result in *NP-hard* problems [5], [6], [18], the character compatibility problem was assumed to be *NP-Complete* as well. This was finally proved independently in 1991 by Steel [29] and Bodlaender, Fellows, and Warnow [1].

Until 1990, the only progress on algorithms for the character compatibility problem was to show that binary character compatibility [27] and compatibility of two characters at a time [13], [17] could be solved in polynomial time. An important theoretical breakthrough came in 1974, when Buneman proved a beautiful result [3] showing that the problem reduced in polynomial time to the graph-theoretic *Triangulating Colored Graphs Problem.* These problems were shown polynomially equivalent by Kannan and Warnow in [21] in 1990. This equivalence was used in [1] to show that the character compatibility problem was *NP-Complete*, so that no polynomial time algorithm for the character compatibility problem will exist unless $P = NP$.

In the biological literature, the first algorithms presented to determine compatibility of two characters were given in 1975 [13], [17] and only proved correct two years later by Estabrook and McMorris in [14]. Algorithms for three characters at a time were discovered first by Kannan and Warnow [21] and later by Bodlaender and Kloks [2] and Idury and Schaffer [20]. Each of these algorithm used linear time, but [21] required quadratic space, while the others find linear space implementations. The only algorithm for the character compatibility problem that runs in polynomial time when the number of characters is bounded by a constant is the algorithm of McMorris, Warnow, and Wimer [28] that achieves a running time of $O(r^{k+1}k^{k+1} + nk^2)$. Since this grows exponentially in the number of characters, it is primarily of theoretical interest.

Characters derived from molecular data (such as DNA sequences) are associated with the columns of a multiple alignment and thus have the restriction that the number of states achieved by any particular character is limited to a small constant. In the case of DNA or RNA sequences, this number is four, while for amino-acid sequences, the number is 20. As we will see, by restricting the number of states, we are able to obtain efficient algorithms for constructing perfect phylogenies. (If they exist, of course!)

The first case of this type that was handled was for binary characters. Polynomial time algorithms for binary character compatibility were found by several authors with the best being the $O(nk)$ algorithm of Gusfield [19], for testing compatibility of k characters on n species. This is optimal, since the input requires $O(nk)$ space, and each entry must be checked. Finally, an $O(nk^2)$ algorithm for compatibility of three-state characters was found by Dress and Steel [8].

In this paper we present two new results: an $O(\min(nk^2, n^2k))$ algorithm for the trinary character compatibility problem, and an $O(n^2k)$ algorithm for the quaternary character compatibility problem.

3. Basic concepts. We describe in this section the basic concepts which are used in our techniques for determining character compatibility for four-state characters. These concepts form the basis of our divide-and-conquer algorithms.

3.1. Canonical labeling of phylogenies. Let P be a proposed phylogeny for S in which only the leaves are assigned labels. To determine whether we can label the internal nodes in P so that the phylogeny is *perfect*, we apply a modification of Fitch's algorithm [16] for finding the minimum size assignment of labels to internal nodes. The difference is that we will permit an internal node to take on a *dummy* state, if its state for a character is not constrained by values at the leaves. The technique we will describe will simultaneously set the labels at the internal nodes and determine whether a labeling that is convex for every character is possible.

Let v be an internal node of P that is not yet labeled. We can define $\alpha(v)$ for every character $\alpha \in C$ by using the following rule: $\alpha(v) = \alpha(x)$ if there exist leaves x and y in different components of $P - \{v\}$ such that $\alpha(x) = \alpha(y)$. If for some internal node v we do not have such a pair of leaves, and thus for all leaves x and y in different components of $P - \{v\}$ we have $\alpha(x) \neq \alpha(y)$, we let $\alpha(v)$ remain "unset," i.e., it takes on the dummy state, α^*. The labeling we have described thus is determined entirely by the topology and the leaf labels. We call this the *canonical* labeling.

To determine if P with the canonical labeling at the nodes (as described above) is perfect, it suffices, clearly, that for every node v, we do not have a conflict in how to set $\alpha(v)$ for any character α. That is, if $P - \{v\}$ has components $P_1, P_2, \ldots, P_r$, we do not have $s_1 \in P_i$, $s_2 \in P_j$, $t_1 \in P_l$, and $t_2 \in P_m$ such that $i \neq j$ and $l \neq m$ and $\alpha(s_1) = \alpha(s_2) \neq \alpha(t_1) = \alpha(t_2)$. Thus, we can determine whether a phylogeny is perfect by knowing the topology and examining the labels at the leaves.

We can assume that every perfect phylogeny P is given the canonical labeling.

Two adjacent nodes v, w in P can be identified if, under the canonical labeling, for each $\alpha \in C$, whenever $\alpha(v) \neq \alpha(w)$, " $*$ " $\in \{\alpha(v), \alpha(w)\}$. In other words, if two nodes disagree for a character only when one is given the dummy state, then we can identify the two nodes. We will sometimes use this flexibility in a phylogeny and thus change the topology of a perfect phylogeny.

We make the following definitions.

DEFINITION 1. *Let P be a perfect phylogeny such that, using the canonical labeling, every two adjacent nodes v and w* cannot *be identified. Then P is said to be* rigid.

3.2. Tree-structures. Let P be a (possibly perfect) phylogeny for S, and consider a specific character, α. The α-tree-structure for P is formed by repeatedly contracting edges connecting nodes that have the same (nondummy) state for α. This tree does not depend on the sequence of edge-contractions and is thus well defined. We are interested in the set of tree-structures that can arise from perfect phylogenies and hence make the following definition.

DEFINITION 2. *A tree-structure for α that can be obtained from some perfect phylogeny will be called a* realizable tree-structure *for α.*

Note that when a phylogeny is perfect, the resultant tree-structure for α has exactly one node for each state of α.

For quaternary characters, the set of possible tree-structures is shown in Fig. 1.

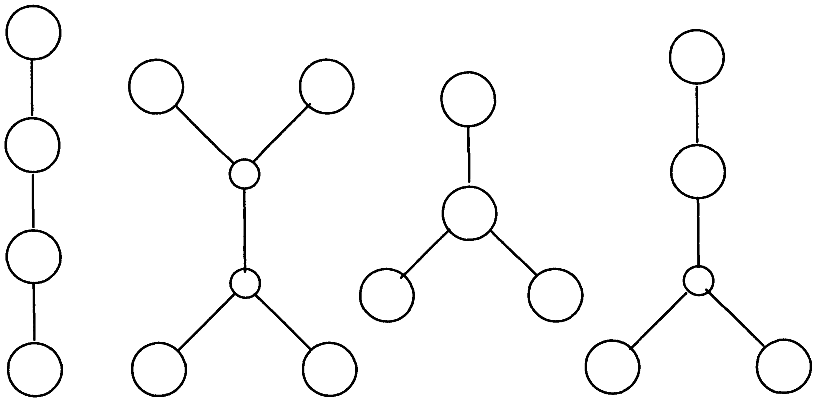


FIG. 1. *Tree structures for a quaternary character.*

We first make the following definition.

DEFINITION 3. *We will say that a perfect phylogeny P is in* reduced form *if it satisfies the following conditions*:

(i) *every internal node of P has degree at least three, and*

(ii) *P has $|S|$ leaves, each labeled by a distinct element of S.*

Clearly, every perfect phylogeny can be put in reduced form and then labeled using the canonical labeling. Thus, in searching for a realizable tree-structure, it will suffice to look for a tree-structure arising from such a perfect phylogeny. We make the following definition:

DEFINITION 4. *A tree-structure that arises from a perfect phylogeny in reduced form and canonically labeled is called a* realizable elementary *tree-structure.*

It is not hard to prove that a phylogeny in reduced form is defined by its set of tree-structures (one for each character in C). It can also be noted that when a tree-structure is derived from a canonically labeled phylogeny in reduced form, then a node with a dummy state has degree at least three in the tree structure.

The tree-structure of a perfect phylogeny for a character α is thus node labeled by α states. We will define some notation that will be used in representing tree-structures as well as in the algorithm statement.

DEFINITION 5. *The denote by α_i the set $\{s \in S : \alpha(s) = i\}$.*

We will use a slightly nonstandard definition of "adjacency" in a tree-structure.

DEFINITION 6. *We will say that two states α_i and α_j are adjacent in a tree-structure T if α_i and α_j are connected by an edge or the path from α_i to α_j in T goes only through dummy nodes.*

This definition suits the usage made of the concept of adjacency in this paper. However, when we say that α_i is *between* α_j and α_k, we will mean that the path from α_j to α_k goes through α_i.

We call a set *big* if it contains at least two elements. Thus α_i is big if $|\alpha_i| \geq 2$. We then make the following definition:

DEFINITION 7. *A character is* informative *if it has at least two* big *states.*

Using this definition, we can state the following:

LEMMA 3.1. *Let C be a set of characters for a set S of species, and suppose that the character α is not informative. Then C is compatible on S if and only if $C - \{\alpha\}$ is compatible on S.*

Proof. One direction is obvious. Now suppose that α is a noninformative character (i.e., it has only one state, α_1, which is big), and suppose that $C - \alpha$ is compatible on S. Let P be a perfect phylogeny for S with respect to the characters in $C - \alpha$. The species in S each occupy a leaf and are labeled by a vector of character states of length $|C| - 1$ (one position per character in $C - \{\alpha\}$).

We can now set the α-state for each node in P, both internal and external, in such a way that the resulting node-labeled tree will be a perfect phylogeny for S with respect to the character set C. The internal nodes will all have state α_1, while the external nodes (leaves) will have their α states set according to the "true" character state information. It is easy to see that this is a perfect phylogeny for the enlarged character set. □

We will show that if we can find a realizable tree structure in some manner (for example by consulting an oracle) for an *informative* character, then we can construct subproblems of strictly smaller size so that the two subproblems have perfect phylogenies if and only if the original instance had a perfect phylogeny. More precisely, given a realizable tree-structure for an informative character, we can construct what we call a *legal partition* of the set S, and from this partition, we can define subproblems. We now define these partitions.

3.3. Legal partitions.

Let S be the set of species and C the set of characters.

DEFINITION 8. *Given a phylogeny P, we will let $\mathcal{L}(P)$ indicate the leaf set of P.*

DEFINITION 9. *A partition of S into S_1 and S_2, with $n - 2 \geq |S_1|, |S_2| \geq 2$, is called a* legal partition *if whenever C is compatible on S there is a perfect phylogeny P with the following property: P contains an edge e whose removal breaks up P into subtrees P_1 and P_2 with $\mathcal{L}(P_i) = S_i$, $i = 1, 2$.*

When the character set is incompatible, *any* partition of S into two big parts is a legal partition. The reason we require the cardinalities of each side of the partition to be between 2 and $n - 2$ is that by doing so we ensure that the subproblems on which we will recurse are small enough.

3.4. Splitting vectors. We begin by making the following definition.

DEFINITION 10. *Given a partition $S = S_1 \cup S_2$, we define the* splitting vector x *(sometimes also called the splitting species), as follows. Let $\alpha \in C$ be a character. If there exist $s_1 \in S_1$ and $s_2 \in S_2$ such that $\alpha(s_1) = \alpha(s_2)$, then we set $\alpha(x) = \alpha(s_1)$; otherwise we set $\alpha(x) =$ "$*$".*

Thus, we define the vector x using exactly the technique we used for labeling the internal nodes of a perfect phylogeny. We will refer to x as the *splitting vector* defined by the partition $S = S_1 \cup S_2$.

Note that the method used in setting x depends *only* on the partition $S = S_1 \cup S_2$ and *not* on the actual perfect phylogeny P yielding that partition. If the method of setting x fails (so that for some character α, there are species $s_1, t_1 \in S_1$ and $s_2, t_2 \in S_2$ with $\alpha(s_1) = \alpha(s_2) \neq \alpha(t_1) = \alpha(t_2)$), then we know immediately that either the character set is not compatible or the partition we have is not a legal partition.

LEMMA 3.2. *Let C be a set of characters on the species set S, and let $S = S_1 \cup S_2$ be a legal partition of S. Let x be the splitting vector defined by the partition $S = S_1 \cup S_2$. Then C is perfectly compatible on S if and only if there exists a perfect phylogeny T for S, C and an edge $e \in E(T)$ such that*

1. *$\mathcal{L}(T_i) = S_i$ for $i = 1, 2$, where $T - \{e\}$ has components T_1 and T_2;*

2. *the labeled phylogeny T' formed by inserting x into the edge e is also a perfect phylogeny for S, C.*

Proof. If C is perfectly compatible on S and if $S = S_1 \cup S_2$ is a legal partition, then the perfect phylogeny T and edge e in T must exist producing the partition. Let x be the splitting vector for $S = S_1 \cup S_2$. If we insert x into e (i.e., *subdivide the edge*), it is easy to see that the convexity property still holds for the new tree. Thus, if C is perfectly compatible with S and $S = S_1 \cup S_2$ is a legal partition, then C is perfectly compatible on $S \cup \{x\}$, where x is the splitting vector defined by the legal partition.

The other direction is trivial to prove. □

3.5. Deriving subproblems. We now state the lemma that permits us to use a legal partition of the species.

LEMMA 3.3. *Let C be a set of characters defined on the species set S, and let $S = S_1 \cup S_2$ be a legal partition. Let x be the splitting vector defined by the previous lemma. Then C is compatible on $S_1 \cup \{x\}$ and $S_2 \cup \{x\}$ if and only if C is compatible on S.*

Proof. Suppose C is compatible on each of $S_1 \cup \{x\}$ and $S_2 \cup \{x\}$ and that T_1 and T_2 are perfect phylogenies for $S_1 \cup \{x\}$ and $S_2 \cup \{x\}$, respectively. Define the phylogeny T by identifying the leaf for x in T_1 with the leaf for x in T_2. We will show that this phylogeny is also perfect. Suppose that u and v are nodes in T with $\alpha(u) = \alpha(v)$ for some character α, and let w be a node on the path from u to v. We will show that $\alpha(w) = \alpha(u)$.

Since T_1 and T_2 are perfect phylogenies, if u and v are both within the same T_i, then $\alpha(w) = \alpha(u)$. If u is in T_1 and v is in T_2, then by our definition of the splitting vector x, $\alpha(x) = \alpha(u)$. Without loss of generality, assume that w is in T_1. Then w is on the path from u to x, and therefore $\alpha(w) = \alpha(u)$. Thus, T is a perfect phylogeny.

Now suppose that C is compatible on S but not on, say, $S_1 \cup \{x\}$. Since $S = S_1 \cup S_2$ is a legal partition, the fact that C is compatible on S implies that there exists a perfect phylogeny T and an edge e in T such that $T - \{e\}$ is the union of two subtrees T_1 and T_2, where the leaves of T_i are the species in S_i. Furthermore, the way we set the character states for the splitting vector x (to be inserted into the edge e) allows us to enlarge T_1 into a perfect phylogeny for $S_1 \cup \{x\}$: we simply make x a leaf and attach it to T_1 at the point where the edge e was attached to T_1.

Thus, C is also compatible on $S_1 \cup \{x\}$. □

The subproblems will have one more species each than the sizes of the parts of the partition, and this is why we constrained each set S_i to have at most $n - 2$ species. Note that the way we have defined $\alpha(x)$ may result in α having $r + 1$ states on $S \cup \{x\}$, where α had r states on S. When we reduce to subproblems, however, we will only have r states for α in each subproblem. We now show this.

LEMMA 3.4. *Let $S = S_1 \cup S_2$ be a legal partition, and let x be the splitting vector defined by the previous theorem. Then in each subproblem $S_i \cup \{x\}$, the number of states for a character α is bounded by r, where α had r states on S.*

Proof. If $\alpha(x) \neq$ " $*$ ", then it is trivially true. Otherwise, if $\alpha(x) =$ " $*$ ", then the states of α are partitioned by $S_1 \cup S_2$ (i.e., no state is on both sides of the partition). The only way that $S_1 \cup \{x\}$ can have $r + 1$ states is for all r original states of α to be in the S_1 side. But in this case, S_2 is empty, and hence $S = S_1 \cup S_2$ is not a legal partition. □

At this point, all we need to do is show how to find a legal partition for S, given a realizable tree-structure.

LEMMA 3.5. *Given a realizable tree-structure T for an informative character α, we can find a legal partition of S.*

Proof. The removal of any edge from T defines a partition that is legal, provided that the parts of the partition have at least two and no more than $n - 2$ species each. We will show that T has at least one edge e whose parts satisfy this constraint.

Since α is informative, α must have at least two big states. Let e be an edge in T such that each subtree of $T - \{e\}$ has at least one big state. Then each component of $T - \{e\}$ has at most $n - 2$ species, and thus we have a legal partition. □

We will call this kind of legal partition of S a *legal partition of α* since members of the same α-state belong to the same part of the partition. Note that this implies a constructive algorithm for reducing to subproblems, given a realizable tree-structure.

3.6. The top-down structure of the algorithm. A top-down description of the algorithm is then as follows:

```
Algorithm Findtree(S)
begin
    if $|S| < 7$ then do exhaustive search
    else
        find a legal partition of $S = S_1 \cup S_2$
        define the splitting vector $s$ (as defined in Section 3.4)
        if both Findtree($S_1 \cup \{s\}$) and Findtree($S_2 \cup \{s\}$) produce trees
          $T_1$ and $T_2$ respectively then
          define $T$ to be the tree formed by identifying the leaves for $s$
          in each tree $T_i$ and return T
          else return No
end of algorithm
```

Note that the splitting vector s can be determined in $O(nk)$ time. The algorithm may determine that no legal partition exists, in which case it does not recurse on subproblems but returns "No."

It is clear that the only difficult part of this algorithm is to find a legal partition of S. We will now describe the techniques we use for finding a legal partition when all the characters have at most three states. The techniques we use for four-state characters involve the same basic ideas but require a much more complicated case analysis.

4. Finding a legal partition from trinary characters. For trinary characters, there are only four possible tree-structures: the three paths, $\alpha_i - \alpha_j - \alpha_k$, for $\{i, j, k\} = \{1, 2, 3\}$, and the star with the dummy node in the center. We will describe a rule-based technique for finding a legal partition of S where all characters have three states. The rules we will present will use character state intersection information to determine which of the four tree-structures could arise from canonically labeled perfect phylogenies in reduced form (i.e., be *elementary realizable tree-structures*). These rules will also be applied for deriving a legal partition for quaternary characters and are thus stated and proved in general terms. We may abuse notation and sometimes indicate the *value* of a species s for the character α by α_i instead of i.

4.1. Rules for deriving tree-structures. We now define two rules for reducing the original set of four possible tree-structures (three paths and one star) to a smaller set of tree-structures, when the characters are trinary.

RULE 1. *Suppose for two characters, α and β, we know that $\alpha_1 \subset \beta_1$, $\alpha_2 \cap \beta_2 \neq \emptyset$, and $\alpha_3 \cap \beta_2 \neq \emptyset$. If the character set is compatible and T is a realizable elementary tree structure for α, then α_1 is not on the path between α_2 and α_3.*

Proof. Suppose that the character set is compatible, and yet the only realizable elementary tree-structures have α_1 on the path between α_2 and α_3. Let T be one perfect phylogeny for the species set, and consider the path P in T from α_2 to α_3. On this path there is at least one, and possibly several, nodes labeled α_1. Attached to each of these nodes are one or more subtrees with species in S having state α_1. Each node v in the path must have β-state β_2, since the α_2 species and α_3 species are in different components of $T - \{v\}$.

Now consider the number of possible subtrees containing species labeled α_1 hanging off nodes labeled α_1 in P. There can be at most one such subtree, since if there were two, the node(s) in P between the two subtrees would have to have state β_1, since all species having state α_1 also have state β_1. Thus, we see that the number of subtrees (containing species in S) must be exactly one. Since P is in reduced form and canonically labeled, the tree-structure arising from P is elementary and α_1 does not lie in the path between α_2 and α_3. □

RULE 2. *Let β and α be characters such that $\alpha_1 \cap \beta_1 \neq \emptyset$, $\alpha_1 \cap \beta_2 \neq \emptyset$, $\alpha_2 \cap \beta_2 \neq \emptyset$, and $\alpha_2 \cap \beta_3 \neq \emptyset$. If T is a realizable elementary tree-structure for α, then α_2 falls between α_1 and α_3 in T.*

The proof of this rule is easy, and is omitted.

4.2. Intersection patterns. To describe the intersection patterns between characters we introduce the following notation. If α is an r-state character then for any character β, its intersection with α is represented by an r-tuple, the ith component of which lists all the β-states that have nonempty intersection with α_i.

As an example, consider two trinary characters α and β on $S = \{1, 2, ..., 8\}$, defined by the following partitions: α partitions S into $\{1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}$, and β partitions S into $\{2, 3, 4, 5\}, \{1, 6, 7\}, \{8\}$. Using the representation above, we determine that α_1 intersects β_1 and β_2, α_2 intersects only β_1, and α_3 intersects β_2 and β_3. Thus, the three-tuple used to describe the intersection pattern is $(\beta_1\beta_2, \beta_1, \beta_2\beta_3)$.

The different intersection patterns are then classified by *type*. Since often we only care about β-states that intersect more than one α-state, we will use symbols x, y, and z to represent possibly empty subsets of β-states that are contained in one α-state. Subsets represented by these letters are mutually disjoint and do not include any explicitly mentioned β-states. Whenever we use the above notation we will bear in mind the symmetries due to relabeling of the α- and β-states.

For trinary characters only a few kinds of intersection patterns are possible. Here we list the types up to symmetry. When the symmetry is broken and we have to distinguish between two characters of the same type, we will refer to *subtypes* of a type. A subtype is defined by a particular rearrangement of the α-states.

Type 1: $(\beta_1 x, \beta_1 y, \beta_1 z)$.

Type 2: (β_1, β_1, x).

Type 3: $(\beta_1 x, \beta_1 y, \beta_3)$.

Type 4: $(\beta_1 x, \beta_1 \beta_2 y, \beta_2 z)$.

Type 5: Any other intersection pattern where α and β are both trinary characters.

Some of these types of intersection immediately allow us to reduce the set of four possible tree-structures (the three paths and one star) to a smaller set. We will therefore define the set Q, which will be used in the algorithm for quaternary character compatibility as well as in the algorithm for trinary character compatibility.

4.3. The set Q of candidates.

DEFINITION 11. *Let $\mathcal{R}_i$ be the set of rules used to remove α tree-structures for i-state character information. Q is the set of tree-structures that are not eliminated by applications of these rules.*

The analysis that we will now make of the implications of the different intersection types will be used to reduce the original set of tree-structures to the (possibly smaller) set Q.

A type-5 intersection can only mean that α and β are themselves incompatible, so that the entire set C is incompatible. A type-4 intersection immediately implies (by an application of Rule 2) that the only possible realizable tree-structure is $\alpha_1 - \alpha_2 - \alpha_3$. If β has a type-3 intersection, then by an application of Rule 1, we know that we can remove the $\alpha_1 - \alpha_3 - \alpha_2$ tree-structure from Q. Type-1 and -2 intersections do not allow us to reduce Q.

Now suppose $Q = \emptyset$. In this case, we know that no tree structure is realizable, so that the character set is incompatible. If Q contains a single tree-structure, then that is the tree-structure we use to reduce to subproblems. Otherwise, Q contains at least two tree-structures, and so we have processed only characters of types 1–3. We will show now that we can handle each of these cases. Before we introduce our technique, we need to make some definitions.

DEFINITION 12. *Let T be an α-tree-structure, e be an edge in T, and $T - \{e\}$ have components T_1 and T_2 with underlying species sets S_1 and S_2, respectively. Then we say that the vector $x \in (Z \cup \{\text{“}*\text{”}\})^k$ is* compatible *with the pair (T, e) if the following is true*:

(i) *If there exists $s_1 \in S_1$ and $s_2 \in S_2$ with $\alpha(s_1) = \alpha(s_2)$ then $\alpha(x) = \alpha(s_1)$.*

(ii) *Else, $\alpha(x)$ is either set to the dummy state “ $*$ ” or is set to i, where either S_1 or S_2 is contained in α_i.*

Thus, a vector is compatible with an edge in a tree if we can insert the vector into that edge without changing the convexity property of the tree.

DEFINITION 13. *Let T be a tree-structure for α, and let $e \in E(T)$. We define $f(T, e) = \{x \in (Z \cup \{\text{“}*\text{”}\})^k : x$ is compatible with (T, e) $\}$.*

That is, $f(T, e)$ consists of all the possible vectors, with dummy states permitted, that can be inserted compatibly into the edge $e \in E(T)$.

DEFINITION 14. *For a tree-structure T for α, we define a* disconnector *to be a minimal set of edges of T whose removal puts each of the α-states into a separate component. If $e_1, \ldots, e_t$ is a disconnector of a tree structure T, a set of species $\mathcal{D} = \{s_1, s_2, \ldots, s_t : s_i \in f(T, e_i), i = 1, \ldots, t\}$ will be termed a* disconnecting species set.

We extend these concepts to the case where we have a collection of tree-structures, Q and state the following theorem.

THEOREM 4.1. *Let C be a set of characters defining a species set S, and assume that C is compatible on S. Let $\alpha \in C$ be a character, and let $\mathcal{T}$ be any set of tree-structures that contains at least one realizable tree-structure. If there exists a set D that is a disconnecting species set for every $T \in \mathcal{T}$, then every tree in $\mathcal{T}$ is realizable.*

Proof. Assume that such a set D exists and that $\mathcal{T}$ contains at least one realizable tree-structure, T^*. Let P^* be a perfect phylogeny for S with tree-structure T^*. Since D is a disconnecting species set for T^*, we can insert the elements of D compatibly into their associated edges of T^*, and hence C is compatible on $S \cup D$.

Now let $T \in \mathcal{T}$ be arbitrarily chosen. If we remove the edges associated with the nodes in D from T, we separate T into r components, each containing one state of α. We can then reattach the deleted edges to each of these components and insert into each such edge the associated element of D. Call these subgraphs $T_1, T_2, \dots, T_r$. Each T_i consists of one α-state and at least one element from D. Every two such subgraphs share at most one element of D. Since C is perfectly compatible on $S \cup D$, there is a perfect phylogeny P_i for each subgraph T_i. We can then combine these perfect phylogenies P_i into a perfect phylogeny for $S \cup D$, in the following way. Each $d \in D$ is a leaf in at least two of these perfect phylogenies P_i. Identify the leaves corresponding to each d. The resultant graph P is acyclic, has the convexity property, and contains S and is therefore a perfect phylogeny for S. Furthermore, it is easy to verify that T is the tree-structure for P, so that T is realizable. □

THEOREM 4.2. *Let C be a set of trinary characters defining a species set S, and let us assume that every character in C is informative. Let α be any character in C, and let Q be the set of candidate tree-structures not eliminated by Rules* 1 *and* 2. *Either some big α-state is a leaf in every $T \in Q$, so that we have a legal partition, or either every tree-structure in Q is realizable or none is.*

Proof. If C is not compatible on S, the theorem trivially follows. Otherwise, assume that C is a set of informative characters that is compatible on S and let $\alpha \in C$ be any character. We will either determine that some big α-state is a leaf in every tree-structure in Q or we will define a disconnecting species set D for Q. Since C is compatible, it must contain at least one realizable tree-structure. The existence of D will then ensure that every tree-structure in Q is realizable, by Theorem 4.1.

The proof continues using a case analysis.

Handling only Type-1 and -2 characters. Consider the case where there have only been characters of Types 1 and 2, so that Q consists of all four possible tree-structures. In this case, we will define a vector A as follows. Each character β has exactly one state, β_1, which intersects more than one α-state; let $\beta(A) = 1$. To set the α-state of A, we use a new dummy state. Note that if P is a perfect phylogeny for S, C and e is an edge in P between two α-states, then A can be inserted into that edge without changing the convexity of P. Thus, $\{A\}$ is a disconnecting species set for Q.

When there are Type-3 characters. The definition of the set D depends on the exact constitution of Q, which can vary when there are Type-3 characters. Each Type-3 character removes one path tree-structure from Q. If there is at least one Type-3 character present and $|Q| > 1$, then we either have removed one or two path tree-structures from Q.

If we have removed two path tree-structures, then we have reduced Q to consisting of one path and the star. Without loss of generality, assume that the only path tree-structure in Q is $\alpha_1 - \alpha_2 - \alpha_3$. Then α_1 and α_3 are *leaf-states* in every tree-structure in Q. Since α has at least two big states, without loss of generality, we can assume α_1 is big. Thus, α_1 is a big state that is a leaf in every tree-structure in Q.

If Q contains two paths, say $\alpha_1 - \alpha_2 - \alpha_3$ and $\alpha_3 - \alpha_1 - \alpha_2$, then all Type-3 characters have had the same intersection subtype (that is, exactly as indicated above). Note that for every tree-structure in Q, α_1 and α_2 are adjacent and α_3 is a leaf. If α_3 is a big state, then we are done. Otherwise, if $|\alpha_3| = 1$, we will need to define two vectors A and B so that $D = \{A, B\}$ is a disconnecting species set.

The node A will be inserted between α_3 and $\alpha_1 \cup \alpha_2$ and the node B will be inserted between α_1 and α_2 in each tree-structure. Note that setting $A = x$, where $\alpha_3 = \{x\}$ will always work. To set $\beta(B)$, note that since every character has type 1, 2, or 3, β has exactly one state in common between α_1 and α_2. Let $\beta(B)$ be that common state. It can then be checked that the nodes A and B can be inserted into each candidate tree-structure in Q so as to preserve convexity. Thus, D is a disconnecting species set, so that every tree-structure in Q is realizable. □

We thus have the following.

THEOREM 4.3. *Given a set of three-state characters, we can in $O(nk)$ time either find a legal partition of S or determine that the set of characters is incompatible.*

Proof. We can compute the pairwise intersections of α with every other character β in $O(nk)$ time and apply Rules 1 and 2, thus reducing the set of four possible tree-structures to a (possibly smaller) set Q. If this set Q is empty, then the characters are incompatible. Otherwise, let T be any tree-structure in Q. By Theorem 4.2, T is realizable, and by Lemma 3.5, we can find a legal partition of S given T. □

Using the same top-down structure as described for the quaternary character compatibility problem in §3.6, we now have an algorithm for trinary character compatibility. We now analyze the complexity of this algorithm.

THEOREM 4.4. *We can determine compatibility of k trinary characters on n species in $O(nk \min\{n, 2k\})$ time.*

Proof. It suffices to show that we recurse at most $\min\{n, k\}$ times, since each recursion costs us only $O(nk)$ time. Having computed a legal partition of $S = S_1 \cup S_2$, we note that the number of species always goes down by at least one in each part, so that the number of iterations is at most n. Now consider what happens to a character as we reduce to solving subproblems. If it becomes uninformative on one side, it is eliminated from the subproblem. Furthermore, if the character only has two states on a subproblem, the reduction to subproblems takes $O(n)$ time instead of $O(nk)$ time. Thus, a character continues to contribute to the cost of its subproblem only by being informative and having at least three states. If a character remains informative and has at least three states in one subproblem, then in the other subproblem it is uninformative. Thus, the total number of characters present between the two subproblems goes down by one at each iteration. Therefore, the number of iterations is $\min\{n, 2k\}$, and the algorithm has running time $O(nk \min\{n, 2k\})$. □

4.4. Application to quaternary character compatibility. The algorithm given above for the compatibility of trinary characters has an important extension to quaternary character compatibility. Suppose we are given a set C of quaternary characters (i.e., characters with number of states bounded by four), and suppose that $\alpha \in C$ is actually a trinary character. Then for any quaternary character β (which is compatible with α), its intersection pattern with α either belongs to one of the four types described or is of the form $(\beta_1\beta_2, \beta_1, \beta_3\beta_4)$, which we will denote as Type 6. Thus, any other pattern will indicate pairwise incompatibility of α and β.

If all characters β intersect α with a pattern that belongs to one of the first four types, then we can obtain a legal partition of α using the trinary character algorithm. Suppose, however, that there is a β of Type 6. Then looking at α's intersection with respect to β we note that α has the pattern $(\alpha_1\alpha_2, \alpha_1, \alpha_3, \alpha_3)$. When the intersection pattern of α with β is of the above form (where one state of α is shared by two states of β and another state of α is shared by the other two states of β and no other α states are shared), α is said to induce a *matching* on β. In §5.4 we describe how a legal partition may be found in the presence of matchings. It will be seen that matchings can be handled even in the presence of some characters with fewer than four states. Thus, we will assume (without loss of generality) that all characters are quaternary, since the presence of even one informative character with at most three states can be handled using simpler techniques.

5. The quaternary character compatibility algorithm. We have described in detail in the previous section how we solved the trinary character compatibility problem in polynomial time. We will describe in this section a more complicated case analysis that permits us to solve the quaternary character compatibility problem in polynomial time as well. The algorithm we give for quaternary characters has much the same structure as the algorithm for trinary character compatibility, in that it operates by finding a realizable tree-structure for a character. In §5.1 we state the third rule for reducing the set of tree-structures to a smaller set of *candidate* tree-structures, which presumes an intersection pattern called a *matching*. In §5.2 we describe a new technique (similar to the technique given in Theorem 1) for finding a legal partition from the set Q of candidates. In §5.3 we define the character intersection types and show how applications of Rules 1–3 permit us to derive information about Q from quaternary characters. In §5.4 we show how to derive a legal partition in the presence of a matching pair. In §5.5 we show how to derive a legal partition when no character β induces a matching on the base character. The analysis of the running time is presented in §5.6.

5.1. The third rule. For the trinary character compatibility problem, we defined two rules that allowed us to reduce the set of possible tree-structures to a smaller set, Q, and from Q we either were able to find a big α-state that was a leaf in every tree-structure in Q or to use any tree-structure in Q as the basis of a legal partition. The case for quaternary characters is a bit more difficult. In addition to the two rules described earlier we need one extra rule, which concerns a different type of intersection pattern. However, in addition, the determination of a legal partition from the set Q of tree-structures that are not eliminated by any of these *three* rules is much more complicated. This complication is due in part to the fact that the intersection patterns can be much more involved, due to the larger number of states. Thus, we will use a great deal of case analysis to determine a legal partition from Q. Still, the general approach is similar, in that we derive a legal partition of S by fixing a base character α and then looking at intersections of α's states with the states of other characters.

We begin by describing the third rule we need.

RULE 3. *Let β and α be characters such that $\beta_1 \cap \alpha_1 \neq \emptyset$, $\beta_1 \cap \alpha_2 \neq \emptyset$, $\beta_2 \cap \alpha_3 \neq \emptyset$, and $\beta_2 \cap \alpha_4 \neq \emptyset$. If T is a realizable elementary tree-structure other than a star, then T contains an edge e such that $T - \{e\}$ has components T_1 and T_2 with $V(T_1) = \{\alpha_1, \alpha_2\}$ and $V(T_2) = \{\alpha_3, \alpha_4\}$.*

Proof. If T is a realizable tree-structure that is not a star, then it contains an edge e that is not incident to any leaf. Let P be a perfect phylogeny that has the T tree-structure for α, and let $e' = (u, v)$ be the edge in P corresponding to e. Clearly, $P - \{e'\}$ partitions the α-states. If $P - \{e'\}$ has $\alpha_1 \cup \alpha_2$ on one side and $\alpha_3 \cup \alpha_4$ on the other, then our rule is valid. Otherwise, without loss of generality, let $\alpha_1 \cup \alpha_3$ be on one side and $\alpha_2 \cup \alpha_4$ be on the other. How do we label $\beta(u)$? Since α_1 is on one side and α_2 is on the other, $\beta(u) = 1$. On the other hand, α_3 is on one side and α_4 is on the other, so that $\beta(u) = 2$. This yields a contradiction, so that e' must separate $\alpha_1 \cup \alpha_2$ from $\alpha_3 \cup \alpha_4$. Therefore, $T - \{e\}$ must have components $\alpha_1 \cup \alpha_2$ and $\alpha_3 \cup \alpha_4$. □

DEFINITION 15. *A character β having the above intersection pattern with α is said to induce a* matching *on α. The pair (α, β) is referred to as a matching pair.*

We can now make some observations.

LEMMA 5.1. *Let C be a set of quaternary characters, S a set of species defined in terms of C, and α a character in C. Suppose that Q is the set of candidate tree-structures for α derived* *from C by applications of Rules* 1, 2, *and* 3. *If the set C is compatible, then Q contains a realizable tree-structure.*

Proof. If C is a compatible set of characters on S, then there is a perfect phylogeny P for S. Without loss of generality, P can be assumed to be canonically labeled and in reduced form. Let $\alpha \in C$ be a character, and let T be the tree-structure for P with respect to α. Then T is an elementary realizable tree-structure and hence will not be eliminated by any of the rules above. Thus $T \in Q$. □

5.2. Finding a legal partition from Q. Our algorithm starts by picking a *base character*, α, arbitrarily and finding the set Q of candidate tree structures for α by applications of Rules 1–3. If $|Q| = 0$, by Lemma 5.1, the character set C is incompatible on S. Again, if $|Q| = 1$, we know, by Lemma 5.1, that if the characters are compatible, then there is only one possible tree structure for α, and it is the unique element of Q. We can then split the problem into subproblems using that tree-structure. The difficulty is when $|Q| > 1$. In what follows we describe some general techniques for dealing with the case that Q contains more than one candidate tree-structure. Although these techniques cover most situations, we will, in some cases, find it necessary to change the base character and find a legal partition on the basis of some other character.

We now generalize the techniques we used for trinary characters, in which we derived a legal partition from the set Q. Recall that for trinary characters, we were able to state that any tree-structure that is not eliminated by Rules 1 and 2 is realizable, provided that the set of characters is compatible. We proved this by proving the existence of a disconnecting species set D, which allowed us to transform any given tree-structure from Q into any other. By contrast, when working with quaternary characters, the existence of a disconnecting species set D cannot be assumed, although we may be able to define such a set of vectors under certain circumstances.

Another technique we use for quaternary characters is described in the following theorem.

THEOREM 5.2. *Suppose s is a vector and $\{e_T \in E(T) : T \in Q\}$ is a collection of edges such that s is compatible with every pair (T, e_T). Suppose there exists $T \in Q$ such that $T - \{e_T\}$ has components T_1 and T_2 whose underlying species sets S_1 and S_2 are both big. Then $S = S_1 \cup S_2$ is a legal partition of the original species set.*

The proof of the above theorem follows along the same lines as the proof of Theorem 4.1.

The above two techniques try to find similarities among the tree-structures in Q and exploit this similarity to find a legal partition. In the case where we have a character β inducing a matching on our base character α, such similarities are hard to find. For example, if the intersection of β with α is given by $(\beta_1\beta_2, \beta_1, \beta_3\beta_4, \beta_4)$, then the possible tree structures for α include stars centered at α_1 and α_3 and the four paths with α_1 and α_2 on one side and α_3 and α_4 on the other. It is hard to find splitting vectors that work for both of these types of realizable tree-structures. To handle matchings we introduce a new technique.

The technique we use involves a concept we call a *rigid tree-structure*. Certain phylogenies are flexible in the sense that the labeling at some adjacent internal nodes are compatible and the nodes can therefore be identified, so that the topology changes, and hence also the tree-structure(s). Thus, it is possible to move from one tree-structure to another, and under the proper conditions, we may be able to know that certain tree-structures may be removed from Q by the above process. We discuss the details of this technique in the subsection on handling matchings.

5.3. Analysis of the character intersection information. Our divide-and-conquer algorithm solves the perfect phylogeny problem by exhaustive search when there are at most six species. In practice, this can be done more efficiently by considering a partition of S into two sets containing 2 and $k-2$ species (assuming $|S| = k \leq 6$), defining the splitting vector for this partition, and checking that the two subproblems created both have solutions.

We first preprocess the data so as to remove all noninformative characters. We then assume that the (reduced) character set contains at least one quaternary character, since if it consists of entirely three-state characters, we will use the simpler algorithm described in §4.

We pick a character α arbitrarily from C and call it the *base* character. We then compute the intersections of the α-states with every state of every remaining character.

We list (up to symmetry) the types of intersections that other characters can have with α assuming that they are compatible with α. Thus, we have eliminated all intersection patterns of α with β that indicate that α and β are pairwise incompatible, so that the entire set C is incompatible. Any β whose intersection does not fit any of the following eight types will thus imply incompatibility of the character set. As before we will sometimes refer to subtypes of a type.

As explained before, we will represent the intersection of a character β with a character α by a four-tuple, the ith component of which lists all the β-states that have nonempty intersection with α_i, for $i = 1, \ldots, 4$. We indicate pairwise disjoint (possibly empty) subsets of β-states by lower-case letters (w, x, y, and z), and we use a "." in a component of the four-tuple to indicate that we do not care about the β-states in that component even if they cross α-state boundaries.

Intersection Types of Characters.

1. $(\beta_1 x, \beta_1\beta_2 y, \beta_2\beta_3, \beta_3)$ or $(\beta_1\beta_2\beta_3 x, \beta_1 y, \beta_2 z, \beta_3 w)$.
2. $(\beta_1 w, \beta_1 x, \beta_2 y, \beta_2 z)$, $(\beta_1\beta_2, \beta_1, \beta_1, \beta_3\beta_4)$, or $(\beta_1, \beta_1\beta_2, \beta_2, \beta_3\beta_4)$.
3. $(\beta_1 x, \beta_1\beta_2 y, \beta_2, \beta_4)$.
4. $(\beta_1 w, \beta_1 x, \beta_3, \beta_4)$.
5. $(\beta_1 w, \beta_1\beta_2 x, \beta_2 y, \beta_2 z)$.
6. $(\beta_1 x, \beta_1 y, \beta_1 z, \beta_4)$.
7. $(\beta_1, \beta_1, \beta_2\beta_3, \beta_4)$.
8. $(\beta_1 x, \beta_1 y, \beta_1 z, \beta_1)$ or $(\beta_1, \beta_1, \beta_1, z)$.

There are conclusions which can be drawn from each of these types of intersections, by applying Rules 1–3. These are as follows:

Type 1: When we have an intersection of this type, the tree-structure of α is entirely determined. To make the problem interesting, we will assume that no character β has this type of intersection pattern with α.

Type 2: The first case has β inducing a matching on α, and the last two cases have α inducing a matching on β. In the last two cases, we switch the roles of α and β, since we have special techniques for handling matchings.

Type 3: In this case, we can assume that α_4 is a leaf-state and that α_2 is between α_1 and α_3.

Type 4: We can assume that α_1 and α_2 are adjacent.

Type 5: (α_1, α_2) is an edge in every tree-structure for α, and α_1 is a leaf-state occurring between α_1 and α_2.

Type 6: We can assume that α_4 is a leaf-state.

Type 7: We can assume that α_4 does not fall between α_1 and α_2.

Type 8: This intersection pattern does not permit us to eliminate any tree-structures from the thirty-two possible candidates but does permit us to set the splitting vectors (which will make up our *disconnecting species set*) easily.

The above enumeration also expresses the order in which we will process other characters with respect to α. We first bucket sort all other characters by the type of intersection they have with α and process them starting with Type-1 characters, if any.

Some comments are in order with regard to the algorithmic implementation. If it is determined that α_i is a leaf-state, then we immediately check the cardinality of α_i. If $|\alpha_i| > 1$, then $S = \alpha_i \cup (S - \alpha_i)$ is a legal partition, and we are done.

5.4. Handling matchings. When we have a character β that induces a matching on α we can find a legal partition based either on α or on β. Note that if β induces a matching on α, then α must perforce be a character with four states.

In order to discuss matchings, we will need to analyze the phylogenies that give rise to tree-structures that are stars.

5.4.1. Star tree-structures and their phylogenies. Assume that the only realizable tree-structures for α are stars and that one such tree-structure is an α_1-star, i.e., a star with α_1 as the node at the center. Consider the phylogenetic tree T that yielded this tree-structure. In T the points of attachments P, Q, and R of the α_2, α_3, and α_4 subtrees to the α_1 subtree must be distinct. If two of these points, say P and Q, are not distinct, we can modify T to create a tree T' that yields a path structure for α (Fig. 2).

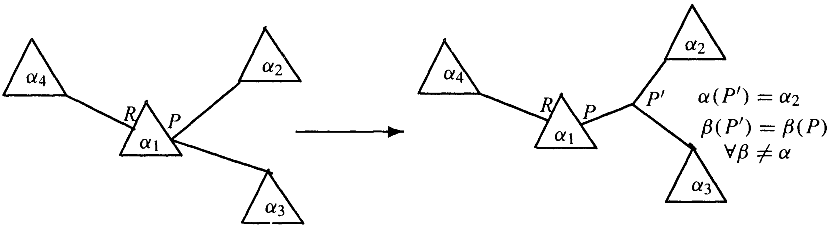


FIG. 2. *Converting a star to a path.*

Furthermore, we may be able to change the character states for these vertices P, Q, and R and still have a perfect phylogeny. If this can be done so that two of these vertices become indistinguishable, then we can transform the star into a path, as before. This can always be done unless we are forbidden from making these changes by the existence of certain characters. For example, we could relabel P and Q unless they differed for some character γ, and we could not change their character state for γ. Thus, $\gamma(P) = \gamma_1 \neq \gamma(Q) = \gamma_2$.

We make the following definition:

DEFINITION. *A star tree-structure for α that cannot be transformed into a path* (*using the techniques described above*) *is called a* rigid star.

If none of the realizable tree-structures for α is a rigid star, then it is clear that α has a realizable tree-structure that is a path (unless, of course, the character set is incompatible). This allows us to reduce to small subproblems, as we showed earlier.

Types of rigid stars and matchings. Let us suppose that we have a matching for α and that we have a rigid star for α, in which α_1 is the center. There are basically two types of perfect phylogenies that yield star tree-structures for α. Figure 3 shows the two different types.

The first star has the points of attachment of α_2, α_3, and α_4 to α_1 all falling on a path. The second star has these points not falling on a single path. We will call the first type a *linear star*, and the second a *nonlinear star.*

We suppose now that β induces a matching on α.

There are, up to symmetry, the following types of matchings:

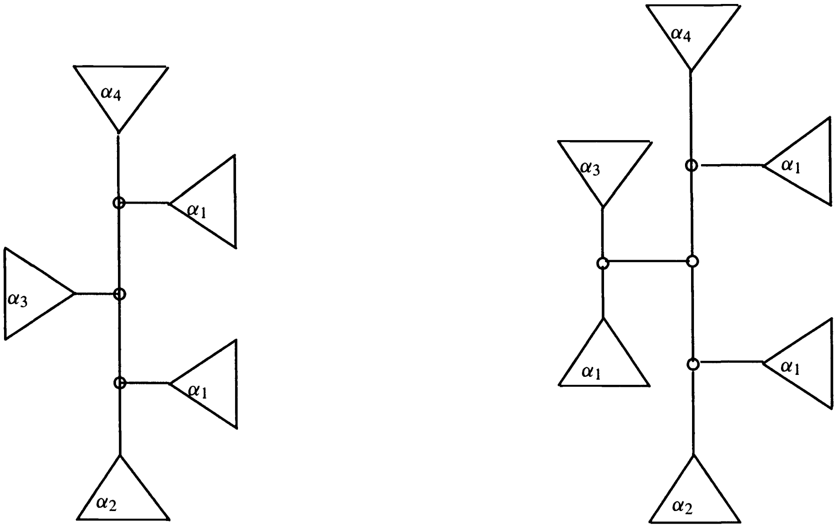


FIG. 3. *Types of stars.*

1. $(\beta_1, \beta_1\beta_2, \beta_3, \beta_3\beta_4)$,
2. $(\beta_1\beta_3, \beta_1\beta_2, \beta_4, \beta_4)$, and
3. $(\beta_1, \beta_1\beta_2\beta_3, \beta_4, \beta_4)$,

We refer to the first two kinds of matchings as *weak* matchings since there are two possible star tree-structures for α in these cases. We will refer to the third kind of matching as a strong matching since in this case the only possible star tree-structure for α is the star with center α_2. (If β has less than four states, the matching is of the form $(\beta_1 x, \beta_1, \beta_2, \beta_2)$.) We can treat this as a special case of a matching of the first kind.

Both linear and nonlinear stars are possible as tree-structures for α if the matching induced on α is strong while only the linear star is possible if the matching is weak. This is because an α-state that is the central state of a nonlinear star must intersect three β-states.

Again this enumeration expresses the order in which the algorithm will process characters that induce matchings.

5.4.2. Matchings of the first kind. Assume that the character β has intersection given by $(\beta_1, \beta_1\beta_2, \beta_3, \beta_3\beta_4)$. If a rigid star exists, by Rule 1, it must be a linear star and have one of α_2 and α_4 for the center. Without loss of generality, let us suppose that a rigid star exists with α_2 as center and the picture looks like Fig. 4. (There could be a potentially different situation if the positions of α_3 and α_4 were interchanged. However, in our argument, these two cases will be completely symmetric. Thus, we can without loss of generality claim that the figure shown is valid.)

Here A is the point of attachment of the α_1 subtree to the α_2 tree, B the point of attachment of the α_3-subtree to the α_2 subtree, and C the point of attachment of the α_4 subtree to the α_2 subtree. The distinctness of B and C and the rigidity of the star implies the existence of a character γ for which C has state γ_1 and B has state γ_2. Further the γ-states of B and C cannot simply be relabeled to be identical, and this implies that γ_1 intersects α_2 and α_4 (and no other α-state) and γ_2 intersects α_3 and one of α_1 and α_2, a fact that we will in the future indicate by the notation

$\gamma_1 : (\alpha_2, \alpha_4)$ and $\gamma_2 : (\alpha_3, \alpha_1 \cup \alpha_2)$.

Note that if $\gamma_2 \cap \alpha_2 \neq \emptyset$, then by Rule 2, the intersection pattern for γ implies that α_2 is between α_3 and α_4. The combined information of β and γ thus rules out all path tree-

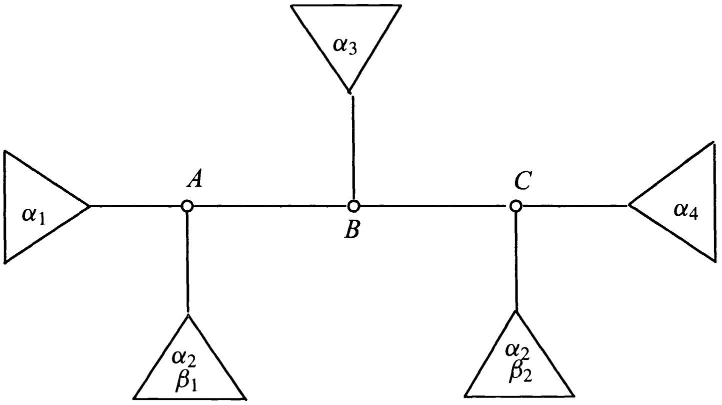


FIG. 4. *A rigid linear α_2 star.*

structures paths as well as the α_4-star, so that Q will contain only the α_2-star. The remaining case, $\gamma_2 : (\alpha_3, \alpha_1)$, requires more careful analysis.

First we note that the intersections of γ and β with α indicate that the only realizable tree-structure for α is a star. If β were a trinary character, then β_2 or β_4 would be empty, so that at most one of the α-states would not be contained inside a single β-state. Thus, Q would be reduced to a single tree-structure. Thus, if β were trinary, we would be able to obtain a legal partition and recurse. If β is truly quaternary, then we have reduced Q to two tree-structures, the α_2-star and the α_4-star.

Consider the intersections of γ with β. If we examine the previous figure, we can see that $\gamma_1 : (\beta_2, (\beta_3 \cup \beta_4))$ and $\gamma_2 : (\beta_1, \beta_3)$. If $\gamma_1 \cap \beta_3 \neq \emptyset$, then the tree-structure for β is forced to be a star with β_3 as the center. Thus, we can assume that $\gamma_1 : (\beta_2, \beta_4)$.

We know at this point that any realizable tree-structure for β must be a star with either β_1 or β_3 as center. If either β_1 or β_3 has a single γ-state, we can immediately determine which of the two stars is realizable. Similarly if either α_2 or α_4 has a single γ-state, we can determine which of the two α-stars is realizable. Thus, the only case left to handle is the following:

Intersections with respect to α:

$$(\beta_1, \beta_1\beta_2, \beta_3, \beta_3\beta_4),$$

$$(\gamma_2, \gamma_1\gamma_3, \gamma_2, \gamma_1\gamma_4).$$

Intersection with respect to β:

$$(\gamma_2\gamma_3, \gamma_1, \gamma_2\gamma_4, \gamma_1).$$

It can be seen that each of $\alpha_1, \alpha_3, \beta_2, \beta_4, \gamma_3, \gamma_4$ forms a leaf in their corresponding tree-structures. Thus if any one of these states is big, we can obtain a legal partition. On the other hand, if each is occupied by a single species, then, in fact, the entire set of species is limited to six species, since every species in S is in one of these states. However, we stated at the beginning that the case of six or fewer species would be handled by exhaustive search. Thus, we can always find a legal partition of a character in this case.

To sum up, whenever β induces a matching of the first kind on α, either the tree structure of α or β is identified as a star with a particular center or a third character γ is found such that there is a legal partition based on a leaf state of one of α, β, or γ. Otherwise, there is no rigid star tree structure for α, and we can assume the legal partition, $(\alpha_1\alpha_2)$, $(\alpha_3\alpha_4)$.

5.4.3. Matchings of the second kind. We assume that there is a character β whose intersection with α is given by $(\beta_1\beta_3, \beta_1\beta_2, \beta_4, \beta_4)$. Suppose there is a realizable rigid star tree-structure for α. Without loss of generality assume that this is an α_2-star. The configuration is the same as Fig. 4. The distinctness of B and C implies the existence of a character γ such that $\gamma_1 : (\alpha_2, \alpha_4)$ and $\gamma_2 : (\alpha_3, \alpha_1 \cup \alpha_2)$. If γ_2 has a nonempty intersection with α_2, the α_2-star is forced as the tree-structure for α. Otherwise, it is clear that α is forced to have a star tree-structure although the center of the star could be either α_1 or α_2. In order for us not to be able to decide which star is the correct one, γ must have at least two of its states intersecting each of α_1 and α_2. This forces the intersection of γ with α to look like : $(\gamma_2\gamma_3, \gamma_1\gamma_4, \gamma_2, \gamma_1)$, a matching of the first kind. Since we look for matchings of the first kind before we look for matchings of the second kind, this case will not occur and thus the tree-structure for α will always be determined.

5.4.4. Strong matchings or matchings of the third kind. Now, suppose we have a matching on α induced by β and that the intersection of α and β is $(\beta_1, \beta_1\beta_2\beta_3, \beta_4, \beta_4)$. Suppose we have a phylogenetic tree T for which the tree-structure on α is a rigid star with α_2 as the center. We have two cases, depending on whether the star is linear or nonlinear.

Case 1: *The rigid star is non-linear.* The tree T must have the structure shown in Fig. 5.

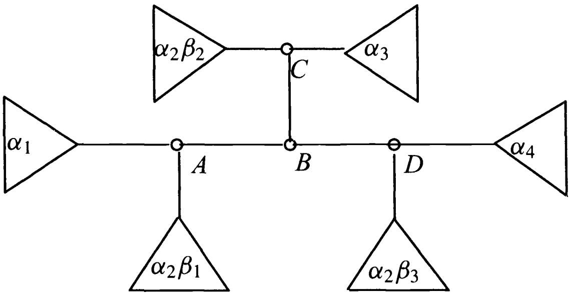


FIG. 5. *Rigid non-linear star.*

The rigidity of T implies that B and C differ for some character γ, since otherwise, we could transform T into a linear star. Thus, we assume that $\gamma(C) = \gamma_1$ and $\gamma(B) = \gamma_2$. It follows that $\gamma_1 : (\beta_4, \beta_2)$ and $\gamma_2 : (\beta_1, \beta_3 \cup \beta_4)$. The distinctness of B and D in turn implies the existence of a character δ such that $\delta_1 : (\beta_3, \beta_4)$ and $\delta_2 : (\beta_1, \beta_2 \cup \beta_4)$. We thus have three possibilities for each of γ_2 and δ_2 and nine possibilities for combinations. In all but two cases, the information gathered from γ and δ allows us to determine the tree-structure for β as a star centered at β_4. The two problem cases are:

(1) $\gamma_1 : (\beta_2, \beta_4), \gamma_2 : (\beta_1, \beta_3), \delta_1 : (\beta_3, \beta_4), \delta_2 : (\beta_1, \beta_2)$.

In this case, it is clear that the tree-structure for β must be a star and the center must be either β_1 or β_4. In order for us not to be able to decide which star is the correct one, neither β_1 nor β_4 can be contained in a single δ-state, and hence we have the following intersection of δ with respect to $\beta : (\delta_2\delta_3, \delta_2, \delta_1, \delta_1\delta_4)$. In this case, however, δ induces a matching of the first kind on β and we switch to β as the base character and process this weak matching.

(2) $\gamma_1 : (\beta_2, \beta_4), \gamma_2 : (\beta_1, \beta_4), \delta_1 : (\beta_3, \beta_4), \delta_2 : (\beta_1, \beta_4)$. It is clear that β_1 must be a leaf-state, and since β_1 is not a singleton, we have a legal partition of β.

Thus, in each of these cases, we were either able to determine a legal partition of β or we reduced to handling a matching of the first kind. Note that without such characters γ and δ, the star would not be rigid, and thus we would have been able to assume that a path was a realizable tree-structure for α, yielding a nontrivial partition of the character α.

Case 2: *The rigid star is linear.* Let us now consider the case of strong matchings, where the rigid star is linear. In this case, we have the same $\alpha - \beta$ intersection as before, and the phylogenetic tree yielding the rigid star must essentially look like Fig. 4 although α_2 now has three β-states instead of two.

As before, we note that the rigidity of the star and the distinctness of B and C imply the existence of a character γ such that $\gamma_1 : (\alpha_2, \alpha_4), \gamma_2 : (\alpha_3, (\alpha_1 \cup \alpha_2))$. Each of the possible cases indicates that a path is not a realizable tree-structure for α, and hence it must be that the tree-structure for α is a star (unless, of course, the character set is incompatible). As we mentioned earlier, since we have a strong matching, only α_2 can be a center of a star. Thus, we know that the tree structure is the star with α_2 as center. This gives us a legal partition of α.

Thus, given any strong matching of β on α, we are either able to find a legal partition of either α or β or we reduce to solving a weak matching.

5.5. When no character induces a matching on α. We assume now that Q is nonempty and contains at least two tree-structures and that we had no matching pairs involving α. This indicates that the only characters we may have seen are of Types 3–8. In the light of our discussion in §4.4, we will also assume that all characters are truly four-state characters (i.e., each character partitions the species set into four nonempty subsets).

We will attempt to define three vectors A, B, and C that form a disconnecting species set for every $T \in Q$. If we succeed, we can divide into subproblems on the basis of any tree-structure in Q, by applying Theorem 4.1 and Lemma 3.5. In what follows we show that in cases where Theorem 4.1 does not apply, we either find a legal partition of α or find a matching pair (β, γ), or discover that the characters are not compatible. Thus in all cases, our algorithm either succeeds in dividing into subproblems or discovers that the character set is incompatible.

Type 3: Suppose that we process a character β of Type 3, with the vector $(\beta_1 x, \beta_1\beta_2 y, \beta_2, \beta_4)$ describing the intersection between α and β. Such a character informs us that α_2 is between α_1 and α_3. Furthermore, we know that α_4 is a leaf-state and that α_2 is never a leaf-state. If α_4 contains more than one species, we have found a legal partition. Otherwise let s be the only species in α_4. After processing all other characters if Q consists of a single tree-structure, we will have found a legal partition. Hence assume that Q has more than one tree-structure. We can name our splitting vectors in this case, with A inserted between α_1 and α_2, B inserted between α_2 and α_3, and C inserted in the edge from α_4. Having identified these splitting vectors, we then set $\beta(A) = \beta_1$, $\beta(B) = \beta_2$, and $\beta(C) = \beta_4$.

For any subsequent character γ, we set the γ-values of A, B, and C as follows: $\gamma(C) = \gamma(s)$. For A if either α_1 or α_2 is covered by a single γ-state, γ_1, we set $\gamma(A) = \gamma_1$. Otherwise, if a γ-state, γ_1, intersects both α_1 and α_2, we set $\gamma(A) = \gamma_1$. Any other intersection pattern for γ that is consistent with the partial α tree-structure implies that α induces a matching on γ and hence this situation would have been handled earlier. Splitting vector B is handled symmetrically to A.

We show why the splitting vectors correctly define a disconnecting species set. Since α_4 is a singleton and a leaf-state in all tree-structures, C is certainly correct.

Now consider vector A. Clearly, if α_1 and α_2 are truly adjacent in a tree-structure (without dummy nodes in between), then A can be inserted compatibly into the edge between α_1 and α_2. There is only one tree-structure T^* not of this form. T^* has a dummy node d, with α_1, α_2, and α_4 connected by edges to d and α_3 a leaf hanging off α_2. We will show that if $|Q| > 1$, then we can identify one of the two edges in the path from α_1 to α_2 in T^* in which we can place A compatibly. This will show that we can place A compatibly into every tree-structure in Q.

If we place A on the edge (α_1, d) in this tree structure, then the only potential problem is a character γ such that γ_1 is shared by α_1 and α_4 and γ_2 covers α_2. Such a character might cause $\gamma(A)$ to be set to γ_2, which would be incorrect. However, if such a character γ exists and $|Q| > 1$, we will show we can place A in the edge (α_2, d). This placement of A will always work unless there is a character δ such that δ_1 is shared by α_2 and α_4 and δ_2 covers α_1. In this case we could set $\delta(A) = \delta_2$, and this would be incorrect. However, note that γ implies that α_2 is not between α_1 and α_4 and that δ implies that α_1 is not between α_2 and α_4. But the combined character information from β, γ, and δ reduces Q to a single possible tree-structure, T^*. Since we have presumed $|Q| > 1$, we can always insert A compatibly into T^* and, hence, into every tree-structure in Q.

Using a symmetric analysis, we show that we can define the vector B and insert it compatibly into every tree-structure in Q, unless $|Q| = 1$.

Having set the splitting vectors A, B, and C, we then apply Theorem 4.1 and Lemma 3.5 to derive a legal partition of S.

Type 4: If the first character β is of type four, then without loss of generality it is of the form $(\beta_1, \beta_1\beta_2, \beta_3, \beta_4)$ and says that α_1 and α_2 are adjacent in every realizable tree-structure for α. If after processing all characters there is a big leaf-state in all tree-structures in Q or if $|Q| = 1$, we can derive a legal partition. We assume that this is not the case. Note that since we presume β to be truly quaternary, the α_2-state is *big*; thus, if the intersection patterns permit us to deduce that α_2 is a leaf in every tree-structure, then we have a legal partition. We will think of A as being inserted on the edge incident to α_1 along the path to α_2. We set $\beta(A) = 1$. We need to define $\gamma(A)$ for every subsequent character γ.

We begin our analysis by noting that in the presence of certain Type-4 characters γ, we will switch to processing β as our base character. If there is a second Type-4 character γ that says that α_3 and α_4 are adjacent, then γ has intersection pattern either $(\gamma_1, \gamma_2, \gamma_3\gamma_4, \gamma_4)$ or $(\gamma_1, \gamma_2, \gamma_4, \gamma_3\gamma_4)$ with α. It is then easy to see that γ induces a matching on β with γ_2 shared by β_1 and β_2 and γ_4 shared by β_3 and β_4. We then switch to β as the base character and process it as in Type 2. Similarly, if γ is a subsequent Type-4 character implying that either α_1 and α_3 or α_1 and α_4 are adjacent, then γ is of Type 3 with respect to β, and we switch to β as the base character and process it as in Type 3. Thus, we only need to concern ourselves with processing Type-4 characters that imply adjacencies between α_1 and α_2, α_2 and α_3, or α_2 and α_4.

We need to show how to set $\gamma(A)$ for characters γ of Type 4 that indicate adjacency of α_1 and α_2, α_2 and α_3, or α_2 and α_4. If γ is a subsequent Type-4 character indicating that α_1 and α_2 are adjacent (by $\gamma_1 \cap \alpha_i \neq \emptyset$, for $i = 1, 2$), we set $\gamma(A) = 1$. If γ is a subsequent Type-4 character and says that α_2 and α_3 or α_2 and α_4 are adjacent, α_1 must be covered by a single γ-state γ_1 and we will set $\gamma(A) = 1$. Thus for every subsequent Type-4 character γ, we can set $\gamma(A)$.

If γ is of Type 5, then there must be a γ-state, γ_1, shared by α_1 and α_2 and we set $\gamma(A) = 1$ (otherwise α_1 and α_2 are *not* adjacent, as implied by β). If γ is of Type 8, we set $\gamma(A) = 1$ where γ_1 is the state of γ that crosses α state boundaries. If γ is of Type 6 and there is a γ-state, γ_1, shared by α_1 and α_2, we set $\gamma(A) = 1$. If not, α_1 must be covered by a single γ-state, γ_2, and we set $\gamma(A) = 2$.

If γ is a Type-7 character, then there is exactly one γ-state, γ_1, that crosses α-state boundaries. If γ_1 has a nonempty intersection with α_1 or α_2, then we set $\gamma(A) = \gamma_1$. Otherwise, one of α_1 or α_2 must be covered by one γ-state, γ_3, and we set $\gamma(A) = \gamma_3$. Clearly, if $\alpha_1 \subset \gamma_1$, this setting will permit us to insert A compatibly into every tree-structure in Q. The only problem arises when intersection of γ with α is given by $(\gamma_1\gamma_2, \gamma_3, \gamma_4, \gamma_4)$. However, in this case γ's intersection with β is given by $(\gamma_1\gamma_2\gamma_3, \gamma_3, \gamma_4, \gamma_4)$ and hence γ induces a matching on β and thus we can process β instead.

Thus we have shown that either we can switch to processing a different character as the base character or that we can set A compatibly with every tree-structure in Q. At this point we check to see if the split produced by A is a legal partition in at least one of the tree-structures in Q. If so, we can apply Theorem 5.2 and use that legal partition. Otherwise α_1 must be a leaf in all of the tree-structures and so presumably a singleton state (otherwise we have a legal partition). Since α must have at least two big states, without loss of generality, let α_3 be a big state. We assume that α_3 is not a leaf in all the tree-structures in Q (since otherwise we can derive a legal partition).

We will now set the splitting vector B. The placement of the vector B into each tree-structure T in Q depends on the topology of T. If T contains an edge e whose removal separates $\{\alpha_1, \alpha_2\}$ from $\{\alpha_3, \alpha_4\}$, then we will insert B into edge e. Otherwise, α_3 is a leaf, and we insert B into the edge incident to α_3. We set $\beta(B) =$ " $*$ " where β^* is a new state of β. We now show how to define $\gamma(B)$ for every subsequent character γ.

Of the Type-4 characters, by a previous analysis we only need to concern ourselves with characters γ that indicate adjacency between α_1 and α_2, α_2 and α_3, or α_2 and α_4, because the other subtypes have intersections with β of Types 2 or 3, permitting us to switch to processing β. On the other hand, since α_3 is presumed to be a big state, we can presume that we do not have characters of Type 4 indicating that α_2 and α_4 are adjacent, since then α_3 is a leaf in every tree-structure and we can derive a legal partition. Handling Type-4 characters γ indicating adjacency between α_2 and α_3 is easy, since then there is a common γ-state γ_1 intersecting both α_2 and α_3, and we can set $\gamma(B) = 1$. The remaining Type-4 character to consider indicates that α_1 and α_2 are adjacent. In this case, α_3 is contained in a single γ-state, γ_1, and we can set $\gamma(B) = 1$.

If γ is of Type 5 or 6 and a γ-state is shared between α_2 and α_3, we can set $\gamma(B)$ to be that γ-state. If not, one of α_2 and α_3 must have been proven to be a leaf and since these are both big states, we will have a legal partition. If γ is of Type 8 and the shared γ-state is γ_1, we simply set $\gamma(B) = \gamma_1$.

Finally if γ is of Type 7 and the γ-state crossing α-state boundaries is γ_1, we will set $\gamma(B) = 1$. It is easy to see that this setting is compatible for tree-structures containing two dummy nodes (so that all α-states are leaves) and any tree-structure containing an edge separating $\{\alpha_1, \alpha_2\}$ from $\{\alpha_3, \alpha_4\}$. The remaining case is where Q contains the α_2-star tree-structure. Again, however, if either α_2 or α_3 is covered by γ_1, then the setting of B is correct. The problem arises if the shared γ-state is between α_1 and α_4. Since in the α_2-star α_2 comes between α_1 and α_4, by Rule 1, we must have α_2 covered by two γ-states, γ_2 and γ_3. In this case α_3 is covered by the single γ-state, γ_4. If such a γ exists, then it says that α_3 is not between α_1 and α_4. The character β says that α_3 is not between α_1 and α_2. Together, these two characters imply that α_3 is a leaf in all the tree-structures in Q. Since α_3 was presumed to be a big state, we can derive a legal partition.

Types 5–8: Suppose all the characters have intersection types between 5 and 8. We will show that we can define a single splitting vector, A, such that one of the following will be true.

(1) A will be compatible with every tree structure in Q other than nonrigid stars, and A will produce a legal partition in at least one of these tree-structures, OR

(2) some big α-state will be a leaf in every tree-structure in Q.

If the first case holds, then Theorem 5.2 tells us that we can use A to derive a legal partition. In the second case, we can derive a legal partition using the big α-state. Thus, in either case, we can derive a legal partition.

The vector we choose will have its states set as follows: If β is a Type-5 character intersecting α, $\beta(A)$ will be the unique β-state that intersects three α-states. For any other character β, $\beta(A)$ will be the only β-state that crosses α-state boundaries.

We will now assume that there is no α-state that is a leaf in every tree-structure in Q and that is also a big state. Using this, we will show that A is compatible with every tree-structure in Q (other than the nonrigid stars). We will look at each type of tree structure in turn.

It is clear that tree-structures with two dummy nodes will not be in Q if there is a Type-5 character. When such tree-structures are in Q (so that all characters have Types 6, 7, or 8), then for every character β, one of the dummy nodes must have the β-state that crosses α-state boundaries. We can assume then that both of the dummy nodes have this β-state since the dummy nodes are adjacent and there is no constraint on the β-state of the other dummy node. Clearly the vector consisting of the common states is a splitting vector whenever the tree-structure has two dummy nodes.

When the tree structure has one dummy node, we will again show that our vector is a valid setting for the position represented by the dummy node. For a Type-5, -6, or -8 character, this is obvious. For a Type-7 character, if the common β-state is shared by the two α-states on one side of the dummy node, we can still set the β-state of the dummy node to be that β-state since it is the only one that crosses α-state boundaries and it is present at a neighboring node to the dummy node.

When the tree structure is a path, we will show that the splitting vector in the middle edge can be set in the way we described. Again for Type-5, Type-6, and Type-8 characters this is obvious. If the character is Type-7 also, it is pretty evident since if the shared β-state does not cross the middle edge, we can still set it to be the β-state of the middle edge because it is the only state on a node incident with the middle edge.

Now consider the case where Q contains a rigid star, T. For the sake of our discussion, we will assume T is an α_1-star. It is clear from our earlier analysis that α_1 will not be a singleton state. Thus, at least one edge in this tree-structure will be between two big states, α_1 and some other α-state. We will prove that either we can insert A compatibly into *every* such edge or we will be able to deduce that either the α_1-star is nonrigid or that some big α-state is a leaf in every tree-structure in Q.

Consider the case where β is a Type-5 character, in which β_1 intersects α_1 and α_2 and β_2 intersects α_1, α_3, and α_4. The intersection pattern then implies that α_2 is a leaf in every tree-structure in Q, so that we can presume α_2 is a singleton state. The β setting for A is then seen to be compatible with the edges (α_1, α_3) and (α_1, α_4), which go between two big states.

If β is a Type-6 character, then without loss of generality β has intersection pattern $(\beta_1 x, \beta_1 y, \beta_1 z, \beta_4)$. This implies that α_4 is always a leaf in every tree-structure in Q. It follows that α_4 is a singleton state (since otherwise we can derive a legal partition). By our construction, then, we will have placed A into an edge between α_1 and one of the remaining two α-states, and for each of these two edges the β-state of A is compatible.

There are two subtypes for Type-8 characters γ; one has a γ-state γ_1 intersecting all the α-states, and the other has one γ-state intersecting three α-states, while the fourth α-state is contained in a single γ-state. In the first case, γ_1 is always compatible with every edge of any tree-structure. In the second case, the fourth α-state is a leaf in every tree-structure in Q and so is small (otherwise we have a legal partition). Thus, in each edge between two big states, the γ setting for A is compatible, when γ has Type 8.

Hence, in each of the cases where β is a Type-5, -6, or -8 character, A can be inserted compatibly into any edge in the α_1-star tree-structure between two big states.

Now, let β be a Type-7 character with β_1 the sole β-state crossing α-states. If α_1 is covered by β_1, then we can insert A into every edge of the α_1-star tree-structure. If α_1 is covered by a single β-state that is not β_1, then we would not have the α_1-star in Q. The only other possibility is that α_1 is covered by two β-states. Without loss of generality, we will assume that β is given by the intersection $(\beta_2\beta_3, \beta_4, \beta_1, \beta_1)$, indicating that α_2 does not fall between α_3 and α_4. We will now show that this star is either nonrigid or that the character that makes this star rigid would rule out enough tree-structures for us to deduce a big leaf-state.

So let us assume that this α_1-star is rigid. Recall that to ensure rigidity of a star we need characters with two states that cross α-state boundaries. Since all the characters are of Types 5–8, the only characters that can ensure rigidity of this star must be Type-5 characters. If there is a perfect phylogeny P realizing the rigid α_1-star, let X, Y, and Z be the points of attachment to the α_1 subtree of the α_2, α_3, and α_4 subtrees, respectively. Since α_1 has only two β-states, X, Y, and Z must lie on a path and one of Y and Z must be an extreme point on the path. Without loss of generality assume that Y is an extreme point. Since the entire path from Y to Z must be labeled by β_1, there can be only two α_1 subtrees hanging off this path, one with β-state β_2 and the other with β-state β_3. Thus Y is adjacent to one of X and Z, and to make sure that Y cannot be identified with this neighbor, we must have a Type-5 character γ that indicates (α_1, α_3) is a true tree edge and α_3 is a leaf. Since β implies α_2 is not between α_3 and α_4, this forces α_2 to be a leaf as well. Thus by assumption α_2 and α_3 are singleton states and α_4 is a big state. We will prove that we can insert A compatibly into the edge (α_4, α_1).

From our previous analysis, we will not have any problem with any character δ having intersection type other than the following: $(\delta_1\delta_2, \delta_3, \delta_3, \delta_4)$. This character, however, implies that α_4 is not between α_2 and α_3. The combined information between β, γ, and δ would then imply that α_4 was also a leaf-state. Since we presume α_4 to be a big state, this would contradict our assumption that no big α-state be a leaf in every tree structure in Q.

Note that each edge we indicated for the insertion of A separated S into two big sets, so that we can apply Theorem 5.2 if A can be inserted compatibly in order to obtain a legal partition. The only tree-structure into which we fail to be able to insert A is the nonrigid α_1-star. If the α_1-star is in Q and is rigid, then we are able to obtain a legal partition of S via other means. Thus, we are able to derive a legal partition of S in this case as well.

We summarize with the following:

THEOREM 5.3. *If α is a character that is not involved in a matching pair, we can find a legal partition of S by analyzing the intersection of other characters with α.*

Putting the above theorem together with our results when α is part of a matching pair, we can state our major theorem.

THEOREM 5.4. *The algorithm either determines that the character set is incompatible or produces a perfect phylogeny.*

To summarize, the algorithm uses a divide-and-conquer strategy to reduce to subproblems, by finding a *legal partition* of the species set S. The legal partition that it finds is based on the kinds of intersections the characters have with a fixed base character α. In the event of a *matching pair*, it uses the techniques described in §5.4. When α is not a member of a matching pair, then the algorithm is able to find a legal partition of α by using the techniques of §5.5.

5.6. Analysis of running time. We assume that the input to the problem is a matrix whose (ij)th entry gives the state of species i for character j. We can construct the sets of species corresponding to each of the character states of each of the characters in $O(nk)$ time. The sets are constructed using a sequence of unions as we examine each species in turn. Since we always union singleton sets to existing sets, the elements of the sets are always at depth one, and hence subsequent FIND operations can be performed in $O(1)$ time. Thus, we can now find the pairwise intersections of the character sets in $O(nk)$ time. The algorithm then picks a character α, makes a linear pass through the characters, and bucket sorts them according to type. This is $O(k)$ time. The separation into subproblems takes $O(n)$ time, and defining the splitting vector takes $O(nk)$ time. We then recurse on the subproblems.

Since each of these subproblems involves less than n species, we have an $O(n^2k)$ algorithm.

The overall running time of the trinary character compatibility algorithm is seen to be $O(n^2k)$, the same as for quaternary character compatibility. This has lead us to conjecture that the techniques we use for determining compatibility of four-state characters can be extended to the general case and will yield an $O(r^{r-2}n^2k)$ algorithm for the case of k characters, n species, and with the restriction to at most r states per character.

6. Comments. Recently [15], Richa Agarwala and David Fernandez-Baca have developed an $O(2^{3r}(nk^3 + k^4))$ algorithm for solving the perfect phylogeny problem when the number of states is bounded by r.

7. Acknowledgments. The authors wish to thank Gene Lawler and Manuel Blum for financial help and encouragement and Chris Meacham, who first introduced us to the character compatibility problem. Special thanks are due to Dan Gusfield, who recommended this special case to us and who went over intermediate drafts of this paper with great care.

REFERENCES

[1] H. Bodlaender, M. Fellows, and T. Warnow, *Two strikes against perfect phylogeny*, Proceedings, International Congress on Automata and Language Processing, 1992.

[2] H. Bodlaender and T. Kloks, *A simple linear time algorithm for triangulating three-colored graphs*, J. Algorithms, to appear.

[3] P. Buneman, *A characterization of rigid circuit graphs*, Discrete Math., 9 (1974), pp. 205–212.

[4] J. Camin and R. Sokal, *A method for deducing branching sequences in phylogeny*, Evolution 19, (1965), pp. 311–326.

[5] W. H. E. Day, *Computationally difficult parsimony problems in phylogenetic systematics*, J. Theoret. Biol., 103 (1983), pp. 429–438.

[6] W. H. E. Day, D. S. Johnson, and D. Sankoff, *The computational complexity of inferring rooted phylogenies by parsimony*, Math. Biosci., 81 (1986), pp. 33–42.

[7] W. H. E. Day and D. Sankoff, *Computational complexity of inferring phylogenies by compatibility*, Syst. Zool., 35 (1986), pp. 224–229.

[8] A. Dress and M. A. Steel, *Convex tree realizations of partitions*, Appl. Math. Lett., 5 (1992), pp. 3–6.

[9] A. W. F. Edwards and L. L. Cavalli-Sforza, *The reconstruction of evolution*, Ann. Human. Genet., 27 (1963), p. 105.

[10] ———, *Reconstruction of evolutionary trees*, in Phenetic and Phylogenetic Classification, V. H. Heywood and J. McNeill, eds., London Systematics Association Publication No. 6, London, 1964, pp. 67–76.

[11] G. F. Estabrook, *Cladistic methodology: a discussion of the theoretical basis for the induction of evolutionary history*, Ann. Rev. Ecol. Syst., 3 (1972), pp. 427–456.

[12] G. F. Estabrook, C. S. Johnson, Jr. and F. R. McMorris, *An idealized concept of the true cladistic character*, Math. Biosci., 23 (1975), pp. 263–272.

[13] G. F. Estabrook and L. Landrum, *A simple test for the possible simultaneous evolutionary divergence of two aminoacid positions*, Taxon., 24 (1975), pp. 609–613.

[14] G. F. Estabrook and F. R. McMorris, *When are two qualitative taxonomic characters compatible?*, J. Math. Biol., 4 (1977), pp. 195–200.

[15] R. Agarwala and D. Fernandez-Baca, *A polynomial time algorithm for the perfect phylogeny problem when the number of character states is fixed*, SIAM J. Comput., 23 (1994), to appear.

[16] W. M. Fitch, *Toward defining the course of evolution: minimum change for a specified tree topology*, Syst. Zool., 20 (1971), pp. 406–416.

[17] ———, *Toward finding the tree of maximum parsimony*, Proc. Eighth International Conference on Numerical Taxonomy, G. F. Estabrook, ed., W. H. Freeman, San Francisco, 1975, pp. 189–230.

[18] L. R. Foulds and R. L. Graham, *The steiner problem in phylogeny is NP-Complete*, Adv. in Appl. Math., 3 (1982), pp. 43–49.

[19] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks, 21 (1991), pp. 19–28.

[20] R. Idury and A. Schaffer, *Triangulating three-colored graphs in linear time and linear space*, SIAM J. Discrete Math., 6 (1993), pp. 289–293.

[21] S. Kannan and T. Warnow, *Triangulating three-colored graphs*, SIAM J. Discrete Math., (1992), pp. 249–258.

[22] A. G. Kluge and J. S. Farris, *Quantitative phyletics and the evolution of anurans*, Syst. Zool., 18 (1969), pp. 1–32.

[23] W. J. LeQuesne, *A method of selection of characters in numerical taxonomy*, Syst. Zool., 18 (1969), pp. 201–205.

[24] ———, *Further studies on the uniquely derived character concept*, Syst. Zool., 21 (1972), pp. 281–288.

[25] W. J. LeQuesne, *The uniquely evolved character concept and its cladistic application*, Syst. Zool., 23 (1974), pp. 513–517.

[26] ———, *The uniquely evolved character concept*, Syst. Zool., 26 (1977), pp. 218–223.

[27] F. R. McMorris, *On the compatibility of binary qualitative taxonomic characters*, Bull. Math. Biol., 39 (1977), pp. 133–138.

[28] F. R. McMorris, T. Warnow, and T. Wimer, *Triangulating vertex colored graphs*, SIAM J. Discrete Math., 7 (1994).

[29] M. A. Steel, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 738–761, August 1994

DYNAMIC PERFECT HASHING: UPPER AND LOWER BOUNDS*

MARTIN DIETZFELBINGER†, ANNA KARLIN‡, KURT MEHLHORN§, FRIEDHELM MEYER AUF DER HEIDE¶, HANS ROHNERT‖, AND ROBERT E. TARJAN**

Abstract. The dynamic dictionary problem is considered: provide an algorithm for storing a dynamic set, allowing the operations insert, delete, and lookup. A dynamic perfect hashing strategy is given: a *randomized* algorithm for the dynamic dictionary problem that takes $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions; it uses space proportional to the size of the set stored. Furthermore, lower bounds for the time complexity of a class of *deterministic* algorithms for the dictionary problem are proved. This class encompasses realistic hashing-based schemes that use linear space. Such algorithms have amortized worst-case time complexity $\Omega(\log n)$ for a sequence of n insertions and lookups; if the worst-case lookup time is restricted to k, then the lower bound becomes $\Omega(k \cdot n^{1/k})$.

Key words. data structures, dictionary problem, hashing, universal hashing, randomized algorithm, lower bound

AMS subject classifications. 68P05, 68P10, 68Q20

1. Introduction. A *dictionary* over a *universe* $U = \{0, 1, \dots, N-1\}$ is a partial function S from U to some set I. The operations *Lookup*(x), *Insert*(x, i), and *Delete*(x) are available on a dictionary S; *Lookup*(x) returns $S(x)$, *Insert*(x, i) adds x to the domain of S and sets $S(x)$ to i, and *Delete*(x) removes x from the domain of S. In the following, the "information field" $S(x)$ associated with the "key" x in the dictionary will be ignored; thus, S is identified with its domain and regarded as a (dynamic) set. There are two major techniques for implementing dictionaries: trees and hashing.

For a static set S (no updates), Fredman, Komlós, and Szemerédi [9] described a hashing technique that achieves linear storage (in n) and constant query time for all N and n, where n is the size of S.

In this paper (§2), we present an extension of their scheme to the dynamic situation, wherein membership queries are processed in constant worst-case time; insertions and deletions are processed in constant expected amortized time; and the storage used at any time is proportional to the number of elements currently stored in the dictionary. The algorithm is randomized; the averaging involved in the analysis is over choices made by the algorithm and not over the sequence of operations.

Besides solutions that use (balanced) search trees, several other approaches to the dynamic dictionary problem have been proposed, some of which lead to expected or average constant time per instruction. Aho and Lee [1] presented a scheme achieving the same time and storage

*Received by the editors January 21, 1991; accepted for publication (in revised form) April 27, 1993.

†Universität Dortmund, Fachbereich Informatik, Lehrstuhl II, 44149 Dortmund, Germany. This author's research was partially supported by Deutsche Forschungsgemeinschaft grant Me 872/1-4.

‡DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301. While visiting Princeton University, this author's research was partially supported by National Science Foundation grants DCR-8605962 and STC88-09648 and Office of Naval Research contract N00014-87-K-0467.

§Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany. This author's research was partially supported by Deutsche Forschungsgemeinschaft grant Me 620/6-1 and ESPRIT-project ALCOM. This author was affiliated with the Universität des Saarlandes when this research was done.

¶Universität–GH–Paderborn, Fachbereich Mathematik/Informatik and Heinz Nixdorf Institut 33095 Paderborn, Germany.

‖Siemens AG, 81370 München, Germany. This author's research was partially supported by Deutsche Forschungsgemeinschaft grant Me 620/6-1 and ESPRIT-project ALCOM. This author was affiliated with the Universität des Saarlandes when this research was done.

**Princeton University, Department of Computer Science, Princeton, New Jersey 08544 and NEC Research Institute. This author's research was partially supported by National Science Foundation grants DCR-8605962 and STC88-09648 and Office of Naval Research contract N00014-87-K-0467.

bounds as our algorithm. However, in order to prove these bounds, they require that the items being inserted be chosen uniformly at random from the universe of possible elements.

Carter and Wegman [4] proposed *universal hashing* as a way of avoiding assumptions on the distribution of input values. This approach works particularly well in combination with the idea of "continuous rehashing" introduced by Brassard and Kannan [3]. In this way an algorithm is obtained that needs linear space and expected constant time for each single instruction. However, for n keys being stored in the dictionary in a scheme of this kind the best upper bound known on the expected *worst-case time* for an instruction (i.e., the length of the longest chain in the resulting hash table with chaining) is $O(\log n/\log\log n)$ (cf. [7], [16]), and it can be argued that it is $\Omega(\log n/\log\log n)$ no matter what universal class is used. In fact, this lower bound even holds in the case of uniform hashing, where one assumes that the hash values for different keys are chosen uniformly at random [11], [15].

In contrast, our algorithm guarantees constant time for each membership query.

When we say that no assumption is made about the sequence of operations, we mean that the sequence is arbitrary but fixed before the algorithm starts running. In essence, all that is needed for the analysis is that the sequence of operations be independent of the random choices made by the algorithm. Thus, we require that the party that chooses the sequence of operations not use any knowledge on these random choices to determine which items to insert in the table.

In the second part of the paper (§§4 and 5), we consider the case that we have to deal with an adversary that knows the random choices made by the algorithm or, equivalently, that the algorithm is deterministic. We prove an $\Omega(\log n)$ lower bound on the amortized worst-case time complexity for any deterministic solution to the dictionary problem which is solely based on hashing and uses only linear space. Furthermore, if we assume the worst-case lookup time to be bounded by k, the amortized worst-case complexity is $\Omega(k\cdot n^{1/k})$.

Remark 1.1. Some of the lower bounds that hold for the model considered in §§4 and 5 are bigger than the $O(\log n)$ worst-case bound guaranteed by balanced search trees. This results from the fact that our model is defined so as to cover only pure hashing strategies. In [14], which was motivated by the first version of the present paper, a lower bound of $\Omega(n\log\log n)$ for n insertions is shown in a stronger lower bound model that encompasses both hashing strategies and search trees. In [17], this lower bound was extended, by different methods, to an even wider class of models.

In §3, some general facts concerning the performance of universal classes of hash functions consisting of polynomials of constant degree or variants thereof are established. These results have proved useful for variations of the scheme presented in this paper, which yield constructions of dynamic dictionaries for parallel and distributed machine models as well as further improvements of the sequential scheme [2], [6], [7], [10].

2. Dynamic perfect hashing. We begin by reviewing the Fredman–Komlós–Szemerédi (FKS) scheme for statically storing a set S of size n. Assume p is prime and $p\geq N$. For $1\leq s\leq p$, let

$$\mathcal{H}_s=\{h:U\rightarrow\{0,\dots,s-1\}\mid\exists a\in\{1,\dots,p-1\}\forall x\in U:h(x)=(ax \bmod p)\bmod s\}.$$

The scheme has two levels. At the top level, a hash function partitions the elements being stored into s sets. The second level consists of a perfect hash function for each of these sets. Specifically, a function h chosen uniformly at random from $\mathcal{H}_s$ is used to partition the set S into s blocks. Let $W_j^h=\{x\in S\mid h(x)=j\}$; the superscript h is omitted when h is understood. Fredman, Komlós, and Szemerédi show that if a function h is chosen from $\mathcal{H}_s$

uniformly at random, then

$$E\left(\sum_{0\le j<s}\binom{|W_j|}{2}\right) \le \frac{n(n-1)}{s} \tag{2.1}$$

(where $E(X)$ denotes the expectation of the random variable X) and consequently that

$$\Pr\left(\sum_{0\le j<s}\binom{|W_j|}{2} < \frac{2n(n-1)}{s}\right) \ge \frac{1}{2}. \tag{2.2}$$

Choosing $s = 2(n-1)$, relation (2.2) implies that for at least half of the functions $h \in \mathcal{H}_s$ one has

$$\sum_{0\le j<s}\binom{|W_j|}{2} < n.$$

Such a function is used to partition S into blocks W_j, $0 \le j < s$. For each block W_j one uses relation (2.2) with $s_j = \max\{1, 2|W_j|(|W_j|-1)\}$. It follows that for at least half of the functions $h \in \mathcal{H}_{s_j}$ one has

$$\sum_{0\le l<s_j}\binom{|W_{j,l}|}{2} < 1,$$

where $W_{j,l} = \{x \in W_j \mid h(x) = l\}$, i.e. , $|W_{j,l}| \le 1$ for all l. For each j, therefore, at least half of the functions in $\mathcal{H}_{s_j}$ are injective on W_j. One uses one such function for each W_j. The total space requirement is linear since

$$\sum_{0\le j<s} s_j \le s + 4\cdot\sum_{0\le j<s}\binom{|W_j|}{2} = O(n),$$

by the choice of the hash functions.

For the dynamic case, we use the standard doubling method to deal with the fact that we do not know in advance how big the top-level table or any of the subtables will get.

Suppose that n is the current number of elements stored in the table. The FKS scheme in use will accommodate up to M elements. The value of M will initially be set to $(1+c)\cdot n$, for some c with $0 < c < 1$, and as n changes will never be more than $\frac{1+c}{1-c}\cdot n$. Let $s(M)$, to be specified, be the number of sets into which the top-level hash function is to partition the elements of S. The function h will be a random element of $\mathcal{H}_{s(M)}$. Thus, the set S is partitioned by h into the subsets $W_j = \{x \in S \mid h(x) = j\}, 0 \le j < s(M)$.

Let T_j be the block of memory used for storing W_j. The amount of space allocated to T_j is s_j, where $s_j = 2m_j(m_j-1)$ and m_j is the maximal size of W_j the current table T_j is meant to manage. The quantity m_j is always at least as big as $|W_j|$ and is at most twice the number of all elements ever mapped to j by the current top-level function h. The subset W_j is resolved within T_j by using a perfect hash function h_j from $\mathcal{H}_{s_j}$. If the value a_j specifies which hash function h_j is being used, then $x \in W_j$ is stored in location $(a_j x \bmod p) \bmod s_j$ of subtable T_j. It will be arranged that the following condition is always satisfied:

$$\sum_{0\le j<s(M)} s_j \le \frac{32M^2}{s(M)} + 4M\ . \tag{2.3}$$

The parameter $s(M)$ will be chosen to be $\Theta(n)$ so that the right-hand side of this equation is $O(n)$. We will see that this guarantees that the total space used is linear in the number of elements currently stored in the table.

The algorithm can be specified more precisely as described in the program given in Figs. 1 and 2. The variable *count* keeps track of the number of updates performed in the hash table of the present size M. From time to time it becomes necessary to restructure the whole table. This is the case when *count* reaches M or when condition (2.3) becomes wrong. In both cases, we start a *new phase*, resetting M to the new value $(1+c)\cdot n$, where n is the number of elements currently stored in the dictionary; the variable *count* is set to n, so that the system is able to perform up to $c \cdot n$ updates before the beginning of the next phase. Deletions are performed by attaching a tag "deleted" to the table entry to be erased; only when a new level-one hash function h or a new hash function h_j for the subtable T_j is chosen, do we drop the elements with a tag "deleted" from T_j. Note that all variables occurring in the procedures (in particular M, $s(M)$, *count*, h, h_j, for $0 \leq j < s(M)$) are global; an exception are the lists L and L_j, for $0 \leq j < s(M)$, which are local to the procedures that use them.

Let us first analyze the space needed by the scheme. By a *phase* we mean the time period during which one level-one hash function h is "in use": a phase starts when some h is chosen and ends when the next level-one function is chosen, either in the same or the subsequent call to *RehashAll*. Phases that only consist of choosing an h to find out that h does not satisfy condition (2.3) are called *degenerate*. During any phase that starts with n keys being stored in the dictionary the number of keys will never drop below $(1-c)\cdot n$, since at most $c \cdot n$ updates are made. Thus, the following lemma is sufficient to prove the claimed space bound.

LEMMA 2.1. *The memory space used during a phase that starts with n keys being stored in the dictionary is $O(n)$.*

Proof. The lemma is obviously true for degenerate phases, since $s(M) = \Theta(n)$. Thus, we assume that a function h is chosen that satisfies condition (2.3) and determine how big the table T has to be to accommodate all versions of all subtables. For $0 \leq j < s(M)$, let $\bar{m}_j$ denote the final capacity of T_j, that is, the value of m_j at the end of the phase, and let $\bar{s}_j = 2\bar{m}_j(\bar{m}_j - 1)$ be the final size of T_j. The previous versions of T_j (if there were any) had capacity $\frac{1}{2}\bar{m}_j, \frac{1}{4}\bar{m}_j, \ldots$. Since, for $l \geq 0$,

$$2 \cdot (2^{-l} \cdot \bar{m}_j)(2^{-l} \cdot \bar{m}_j - 1) \leq 4^{-l} \cdot 2\bar{m}_j(\bar{m}_j - 1) = 4^{-l} \cdot \bar{s}_j,$$

the total number of cells occupied by all versions of all subtables T_j is bounded by

$$\sum_{0 \leq j < s(M)} \sum_{l \geq 0} 4^{-l} \cdot \bar{s}_j = \frac{4}{3} \cdot \sum_{0 \leq j < s(M)} \bar{s}_j \leq \frac{4}{3} \cdot \left(\frac{32M^2}{s(M)} + 4M\right).$$

The last inequality holds since the algorithm makes sure that condition (2.3) remains valid throughout the phase.

The space required by the header table is at most $5s(M)$, since the jth entry of the header table need only contain a pointer to T_j, the variables s_j, b_j, and m_j, and the number a_j that describes the hash function h_j. If we let $s(M) = \frac{8}{15}\sqrt{30} \cdot M$, the space needed by the subtables and the header table taken together is bounded by $\frac{4}{3} \cdot ((32M^2 \cdot 15)/(8\sqrt{30} \cdot M) + 4M) + 5 \cdot \frac{8}{15}\sqrt{30} \cdot M = \frac{16}{3}(\sqrt{30} + 1) \cdot M < 35 \cdot (1+c) \cdot n$, which proves the lemma. □

Now we turn to the time bounds. Note first that membership queries do not interfere with the time analysis, since they are executed in constant time in the worst case. Thus, there is no harm in assuming that there are no membership queries at all. Note further that instructions that are executed in constant time (i.e., deletions in any case and insertions if they do not cause a subtable T_j to be rearranged) can be safely ignored, since they will not invalidate an overall linear time bound. Thus, we only need to worry about the time spent for installing

```
procedure $Insert(x)$;
  $count \leftarrow count + 1$;
  if $count > M$
  then
    $RehashAll(x)$;
  else
    $j \leftarrow h(x)$;
    if position $h_j(x)$ of subtable $T_j$ contains $x$
      then
        if $x$ is marked “deleted” then remove this tag;
      else (* $x$ is new for $W_j$ *)
        $b_j \leftarrow b_j + 1$;
        if $b_j \leq m_j$
          then (* size of $T_j$ sufficient *)
            if position $h_j(x)$ of $T_j$ is empty
              then
                store $x$ in position $h_j(x)$ of $T_j$;
              else
                go through the subtable $T_j$, put all elements
                not marked “deleted” into a list $L_j$, and
                mark all positions of $T_j$ empty;
                append $x$ to list $L_j$; $b_j \leftarrow$ length of $L_j$;
                repeat $h_j \leftarrow$ randomly chosen function in $\mathcal{H}_{s_j}$
                until $h_j$ is injective on the elements of list $L_j$;
                for all $y$ on list $L_j$ store $y$ in position $h_j(y)$ of $T_j$;
          else (* $T_j$ is too small *)
            $m_j \leftarrow 2 \cdot \max\{1, m_j\}$; $s_j \leftarrow 2m_j(m_j - 1)$;
            if condition (2.3) is still satisfied
              then (* double capacity of $T_j$ *)
                allocate new space, namely $s_j$ cells, for new subtable $T_j$;
                go through old subtable $T_j$, put all elements
                not marked “deleted” into a list $L_j$,
                and mark all positions empty;
                append $x$ to list $L_j$; $b_j \leftarrow$ length of $L_j$;
                repeat $h_j \leftarrow$ randomly chosen function in $\mathcal{H}_{s_j}$
                until $h_j$ is injective on the elements of list $L_j$;
                for all $y$ on list $L_j$ store $y$ in position $h_j(y)$ of $T_j$;
              else (* level-1-function $h$ “bad” *)
                $RehashAll(x)$;
```

FIG. 1. *Insertion.*

```
procedure RehashAll($x$);
  (* RehashAll is either called by Insert with a parameter $x \in U$,
    or by Delete or Initialize without parameters. RehashAll builds a
    new table for all elements currently in the table (plus $x$, if given). *)
 go through the whole table $T$, put all elements not tagged "deleted"
 into a list $L$, count them, and mark all positions in $T$ "empty";
 if $x \in U$ then append $x$ to $L$;
 $count \leftarrow$ length of list $L$;
 $M \leftarrow (1 + c) \cdot \max\{count, 4\}$;
 repeat $h \leftarrow$ randomly chosen function in $\mathcal{H}_{s(M)}$;
   for all $j, 0 \leq j < s(M)$, do form a list $L_j$ of all $x \in L$ with $h(x) = j$;
   for all $j, 0 \leq j < s(M)$, do
     $b_j \leftarrow$ length of list $L_j$; $m_j \leftarrow 2 \cdot b_j$; $s_j \leftarrow 2m_j(m_j - 1)$;
 until condition (2.3) is satisfied;
 for all $j, 0 \leq j < s(M)$, do
   allocate space $s_j$ for subtable $T_j$;
   repeat $h_j \leftarrow$ randomly chosen function in $\mathcal{H}_{s_j}$
   until $h_j$ is injective on the elements of list $L_j$;
   for all $x$ on list $L_j$ do store $x$ in position $h_j(x)$ of $T_j$;


procedure Delete($x$);
 $count \leftarrow count + 1$;
 $j \leftarrow h(x)$;
 if position $h_j(x)$ of subtable $T_j$ contains $x$
   then mark $x$ as "deleted"
   else return($x$ is not a member of $S$);
 if $count \geq M$
   then (* start new phase *)
     RehashAll();


procedure Lookup($x$);
 $j \leftarrow h(x)$;
 if position $h_j(x)$ of subtable $T_j$ contains $x$ (not marked "deleted")
   then return("$x$ is a member of $S$")
   else return("$x$ is not a member of $S$");


procedure Initialize;
 $T \leftarrow$ an empty table;
 RehashAll();
```

FIG. 2. *Setup, deletion, lookup, and rehashing.*

new level-one functions at the beginning of a phase (in *RehashAll*) and for constructing new versions of the subtables T_j (in *RehashAll* or in *Insert*).

LEMMA 2.2. *The expected time for a phase that starts with n keys being stored in the dictionary is $O(n)$.*

Proof. Consider the call to *RehashAll* in which the phase starts. Clearing the old table (header table and the subtables) and building up the list L takes time $O(n)$, since by Lemma 2.1 the old table occupies only space $O(n)$. Time linear in n suffices to construct the sublists L_j, to compute the values b_j, m_j, and s_j, $0 \le j < s(M)$, and to compute $\sum_{0\le j<s(M)} s_j$. Thus, if the phase is degenerate, it takes $O(n)$ time in the worst case. In a nondegenerate phase h initially satisfies condition (2.3). By the remarks immediately preceding Lemma 2.2, we only have to estimate the time spent for installing new hash functions h_j for the subtables. Fix some j, and split the phase, as far as T_j is concerned, into subphases, one subphase being defined as a maximal time period in which the capacity m_j and hence the size s_j of T_j have a fixed value. We need the following observation:

Claim. Assume a hash function h_j is chosen for T_j at the beginning or in the course of a subphase. Then the probability that h_j stays in use until the end of the subphase exceeds $\frac{1}{2}$.

Proof of Claim. Let the capacity of T_j during the phase be m_j. Let W_j be the set of keys x contained in the list L_j when h_j is chosen. Let V_j be the set of the first $m_j - |W_j|$ *different* keys x in the sequence of the *Insert* instructions to be executed next that satisfy $h(x) = j$ and do not occur in W_j. Then, by relation (2.2), table size $s_j = 2m_j(m_j - 1)$ (for $m_j \neq 0$) implies that with probability exceeding $\frac{1}{2}$ the elements of $W_j \cup V_j$ will be mapped by h_j to different locations in T_j. If this happens, the way b_j is changed and repeated keys are treated in *Insert* and *Delete* implies that h_j stays in use until b_j grows beyond m_j, that is, until the end of the subphase. This proves the claim.

By the claim, the probability that u or more hash functions h_j are used in a single subphase is at most $2^{-(u-1)}$, and hence the expected number of functions h_j chosen during the subphase is bounded by 2. Thus, the expected cost for installing new hash functions h_j during a subphase in which T_j has size s_j is $O(s_j)$. Exactly as in the proof of Lemma 2.1 we get an overall bound of $O(M) = O(n)$ for the expected time for installing new hash functions h_j, $0 \le j < s(M)$, by summing over all subtable sizes and all j, and using condition (2.3). □

In order to finish the time analysis, we will show in the following two lemmas that there will not be too many phases. Fix some phase (no assumptions are made as to whether the phase will turn out to be degenerate or nondegenerate), and let S be the set of elements stored in the table at the beginning of the phase (whose number is n) together with those that occur in the next $c \cdot n$ update instructions to be executed. (Even though elements may occur repeatedly in these instructions, they appear only once in the set S.) Let $M = (1 + c) \cdot n$; clearly, $|S| \le M$. For $h \in \mathcal{H}_{s(M)}$ chosen at random, define $W_j = \{x \in S \mid h(x) = j\}$, $0 \le j < s(M)$.

LEMMA 2.3. (a) *With probability exceeding $\frac{1}{2}$ we have*

$$\sum_{0\le j<s(M)} 4|W_j|(2|W_j| - 1) < \frac{32M^2}{s(M)} + 4M.$$

(b) *If the inequality in* (a) *is satisfied for the level-one function h chosen at the beginning of the phase, then the phase ends with the variable count reaching M; i.e., the phase comprises $c \cdot n$ updates.*

Proof. (a) In the situation just described, relation (2.2) reads

$$\Pr\left(\sum_{0\le j<s(M)} \binom{|W_j|}{2} \le \frac{2M(M-1)}{s(M)}\right) \ge \frac{1}{2}.$$

The claim then follows by a simple transformation, using the obvious inequality $\sum_{0\le j<s(M)} |W_j| = |S| \le M$.

(b) It is immediate from the way the variables b_j and m_j are initialized in *RehashAll* and updated in *Insert* and from the fact that only keys from S can occur in the phase that $b_j \le |W_j|$, and hence $m_j \le 2|W_j|$, throughout the phase. Since the equality $s_j = 2m_j(m_j-1)$ is preserved by the algorithm, we see that the inequality in (a) entails that relation (2.3) stays valid throughout the phase. □

LEMMA 2.4. *Suppose that RehashAll is called at a time when $n \ge 1$ keys are stored in the dictionary. Then the (expected) time needed until the first call to RehashAll after $c \cdot n$ updates have been performed is $O(n)$.*

Proof. Consider an arbitrary phase (degenerate or nondegenerate) that starts before the next $c \cdot n$ updates have been processed. The number of keys in the table at the beginning of this phase is n', where $(1-c)\cdot n < n' < (1+c)\cdot n$. By Lemma 2.3(a),(b), the probability that during this phase $c \cdot n'$ updates are performed exceeds $\frac{1}{2}$. Since $c \cdot n - |n' - n| \le c \cdot n'$ no matter if $n' < n$ or $n' \ge n$, this means that the probability that this phase extends further than the $c \cdot n$ updates we are considering is at least $\frac{1}{2}$. Thus, the expected number of phases (degenerate or nondegenerate) needed to perform these $c \cdot n$ updates is not more than 2. Each phase occurring starts with n' keys, $(1-c)\cdot n < n' < (1+c)\cdot n$, and takes $O(n')$ steps (expected) by Lemma 2.2. This finishes the proof of Lemma 2.4 and the time analysis. □

Lemmas 2.1–2.4 taken together yield the following result.

THEOREM 2.5. *Dynamic perfect hashing, as described by the algorithm in this section, uses linear space, needs constant time for membership queries, and has $O(1)$ expected amortized insertion and deletion cost.*

Remark 2.6. Obviously, the space bound $35(1+c)n$ proved in 2.1 is not satisfactory from a practical point of view. There are many conceivable ways of reducing the space bound: by varying the parameters fixed in the algorithm, by using slightly different hash functions, or by adapting more involved schemes, e. g., that described in [9], which achieves an $n + o(n)$ space bound in the static case. Most of these variations will increase the bounds on the expected computation time, but this does not necessarily mean that the time requirements observed in practice will grow significantly. Wenzel [20] has implemented a variant of the scheme described above. In his implementation the universe U is $\{0, 1, \ldots, 2^{31}-1\}$; the space requirements are kept small by avoiding the use of subtables if $|W_j|$ is small. He reports that the space requirements of his implementation are comparable to those of balanced trees and that the running time is superior to search trees provided n is moderately large ($n \ge 1000$). We refer the reader to [20] for details.

Remark 2.7. It is intuitively clear, and it can be seen in practice, that the evaluation time for the hash functions is critical for the performance of every implementation of the algorithm. In many installations, the functions from the following alternative class will be faster to evaluate than those from the class $\mathcal{H}_s$ above, since division by powers of 2 can be effected by shifts of binary representations of integers. The class corresponds to the classic multiplicative hashing scheme (see [12, p. 509]) with a randomly chosen multiplier. Assume $U = \{0, \ldots, N-1\}$, where $N = 2^\nu$ is a power of 2, and that $s = 2^\lambda$, for $1 \le \lambda \le \nu$. Let

$$\mathcal{H}'_s = \{h \mid \exists a \in \{1, \ldots, 2^\nu - 1\}, a \text{ odd}, \forall x \in U : h(x) = (ax \bmod 2^\nu) \operatorname{div} 2^{(\nu-\lambda)}\}.$$

In [8] it is shown that the class $\mathcal{H}'_s$ also satisfies inequality (2.1), thus it can be used in the algorithm described above without impairing the performance guarantees given by Theorem 2.5.

3. Higher-order hash functions. In this section we generalize inequality (2.1) from §2 (which originated in [9]) to polynomials of degree larger than 1 and note some consequences of this generalization. These extensions have proved useful since the first version of this paper appeared as [5]; see, e. g., [2], [6], [7], [10]. In order to formulate the result in a slightly more general way than just for polynomials, we recall a definition given originally in [19] and studied further (with varying notation) for example in [15], [16].

DEFINITION 3.1 [19]. *Let $\mathcal{H}$ be a collection of functions h with domain D and range R. Let $c > 0$, and let $k \geq 1$ be an integer. The class $\mathcal{H}$ is called* (c, k)-universal *if for all sequences $x_1, \ldots, x_k$ of different elements of D, all sequences $y_1, \ldots, y_k$ of elements of R, and randomly chosen $h \in \mathcal{H}$*

$$\Pr(h(x_i) = y_i \text{ for } 1 \leq i \leq k) \leq \frac{c}{|R|^k}.$$

(Alternatively, such classes have been called c strongly k universal or $(k)_c$-independent.)

Examples. (a) [19] If F is a finite field, we may let $D = R = F$; then

$$\mathcal{H} = \left\{ h \,\middle|\, \exists a_0, \ldots, a_{k-1} \in F : h(x) = \sum_{0 \leq i < k} a_i x^i \text{ for all } x \in F \right\}$$

is $(1, k)$-universal. This holds since for each sequence of k different arguments in F and k prescribed values there is exactly one polynomial of degree at most $k - 1$ that interpolates through these argument-value pairs.

(b) [15], [19] If $\mathcal{H}$ is (c, k)-universal and $r : R \to R'$ is such that $|r^{-1}(j)| \leq d$ for all j, then the (multi)set $\mathcal{H}' = \{r \circ h \mid h \in \mathcal{H}\}$ is $(\hat{c}, k)$-universal, for $\hat{c} = c \cdot (d|R'|/|R|)^k$.

(c) A direct consequence of (a) and (b): If p is prime and $1 \leq s \leq p$, then for $D = \{0, \ldots, p - 1\}$ and $R = \{0, \ldots, s - 1\}$ the set

$$\mathcal{H}_s^k = \left\{ h_{a_0, \ldots, a_{k-1}} \mid 0 \leq a_0, \ldots, a_{k-1} < p \right\},$$

where

$$h_{a_0, \ldots, a_{k-1}}(x) = \left(\sum_{0 \leq i < k} a_i x^i \bmod p \right) \bmod s, \quad \text{for } x \in D \text{ and } 0 \leq a_0, \ldots, a_{k-1} < p,$$

is (c, k)-universal, for $c = \left(\lceil p/s \rceil \cdot s/p\right)^k \leq (1 + s/p)^k$.

(d) For the finite field $D = R = GF(p^l)$, p prime, $l \geq 1$, we obtain $(1, k)$-universal classes with $|R'| = p^{l'}$, $1 \leq l' \leq l$, by combining (a) with a suitable function $r : R \to R'$. (See [15] for further examples.)

In the following, we assume that D and $R = \{0, \ldots, s - 1\}$ are fixed and that $\mathcal{H}$ is a class of functions from D to R. Let a set $S \subseteq D$ be fixed, $|S| = n$, and let x_0 be an element of $D - S$. For $h \in \mathcal{H}$ and $0 \leq j < s$ we define $B_j^h = \{x \in S \mid h(x) = j\}$ and $b_j^h = |B_j^h|$; further, we define $B_{x_0}^h = \{x \in S \mid h(x) = h(x_0)\}$ and $b_{x_0}^h = |B_{x_0}^h|$. Assume that h is chosen uniformly at random from $\mathcal{H}$. (In the notation, we drop the superscript h.) For arbitrary $z \in \mathbb{R}$, $k \geq 0$, we let $(z)_k$ denote the "falling factorial" $z(z - 1) \cdots (z - k + 1)$.

LEMMA 3.2. *If $\mathcal{H}$ is (c, k)-universal for D and R, then*

(a) $E\big((b_j)_k\big) \leq c \cdot (n)_k / s^k \leq c \cdot (n/s)^k$ *for* $0 \leq j < s$;

(b) $E\big((b_{x_0})_{k-1}\big) \leq c \cdot (n)_{k-1}/s^{k-1} \leq c \cdot (\frac{n}{s})^{k-1}$.

Proof. For $l \geq 1$, let $(S)_l$ denote the set $\{(x_1, \ldots, x_l) \in S^l \mid x_1, \ldots, x_l \text{ different}\}$.

(a) Fix j, and define random variables $X_{x_1, \ldots, x_k}$, *for* $(x_1, \ldots, x_k) \in (S)_k$, by

$$X_{x_1, \ldots, x_k} = \begin{cases} 1 & \text{if } h(x_1) = \cdots = h(x_k) = j, \\ 0 & \text{otherwise.} \end{cases}$$

Then $E(X_{x_1,\dots,x_k}) = \Pr(X_{x_1,\dots,x_k} = 1) \le c/s^k$, since $\mathcal{H}$ is (c,k)-universal. On the other hand, it is clear that

$$(b_j)_k = \sum_{(x_1,\dots,x_k)\in(S)_k} X_{x_1,\dots,x_k}.$$

Consequently,

$$E\big((b_j)_k\big) = \sum_{(x_1,\dots,x_k)\in(S)_k} E(X_{x_1,\dots,x_k}) \le |(S)_k| \cdot \frac{c}{s^k} = (n)_k \cdot \frac{c}{s^k}.$$

(b) The proof is similar to the one given in (a). Define random variables $Y^j_{x_1,\dots,x_{k-1}}$, $(x_1,\dots,x_{k-1}) \in (S)_{k-1}, 0 \le j < s$, by

$$Y^j_{x_1,\dots,x_{k-1}} = \begin{cases} 1 & \text{if } h(x_1) = \cdots = h(x_{k-1}) = h(x_0) = j, \\ 0 & \text{otherwise.} \end{cases}$$

Then $E(Y^j_{x_1,\dots,x_{k-1}}) \le c/s^k$, since $\mathcal{H}$ is (c,k)-universal. Further,

$$(b_{x_0})_{k-1} = \sum_{(x_1,\dots,x_{k-1})\in(S)_{k-1}} \sum_{0\le j<s} Y^j_{x_1,\dots,x_{k-1}}.$$

Taking expected values, we get

$$E\big((b_{x_0})_{k-1}\big) \le |(S)_{k-1}| \cdot s \cdot \frac{c}{s^k} = (n)_{k-1} \cdot \frac{c}{s^{k-1}},$$

as claimed. □

A hash function h is called *l-perfect* for S if $b^h_j \le l$ for all j, $0 \le j < s$, i.e., if no block B^h_j has size exceeding l.

COROLLARY 3.3. *In the situation of Lemma* 3.2, *if we further assume that $s \ge n$, we have*:

(a) $\Pr(h \text{ is } (k-1)\text{-perfect}) \ge 1 - (c/k!) \cdot n \cdot (n/s)^{k-1}$. *In case $s = n^{1+1/(k-1)}$ this probability exceeds* $1 - c/k!$.

(b) $E\left(\sum_{0\le j<s} (b_j)^k\right) \le c_k \cdot n$, *for some constant c_k.* (*Here $(b_j)^k$ is the kth power of b_j.*)

(c) $\Pr\left(\sum_{0\le j<s} (b_j)^k \le 2c_k \cdot n\right) \ge \frac{1}{2}$, *for c_k as in* (b).

Proof. (a) We estimate the probability that h is not $(k-1)$-perfect.

Clearly,

$$\Pr(\exists j : b_j \ge k) = \Pr(\exists j : (b_j)_k \ge k!) \le \sum_{0\le j<s} \Pr((b_j)_k \ge k!).$$

By 3.2(a) and the Markov inequality the last term is bounded above by $s \cdot (c/k!) \cdot (n/s)^k = (c/k!) \cdot n \cdot (n/s)^{k-1}$.

(b) Let $J = \{\, j \mid b_j \le k-1\}$. Since $\sum_{j\in J} b_j \le |S| = n$, it follows from elementary considerations that $\sum_{j\in J} (b_j)^k \le \lfloor \frac{n}{k-1} \rfloor \cdot (k-1)^k + (n - \lfloor \frac{n}{k-1} \rfloor \cdot (k-1))^k \le n \cdot (k-1)^{k-1}$.

We need the following simple fact:

Claim. If $z \ge k$, then $z^k/(z)_k < e^{k-1}$.

Proof of Claim.

$$\frac{z^k}{(z)_k} = \prod_{j=1}^{k-1} \frac{z}{z-j} \le \prod_{j=1}^{k-1} \frac{k}{k-j} = \frac{k^{k-1}}{(k-1)!}$$

$$= \sum_{l=0}^{k-1} \binom{k-1}{l} \cdot \frac{(k-1)^l}{(k-1)!} < \sum_{l=0}^{k-1} \frac{(k-1)^l}{l!} < e^{k-1}.$$

Thus, we may write:

$$E\Big(\sum_{0\le j<s}(b_j)^k\Big) \le E\Big(\sum_{j\in J}(b_j)^k + \sum_{j\notin J}(b_j)^k\Big) \le n\cdot(k-1)^{k-1} + e^{k-1}\cdot E\Big(\sum_{0\le j<s}(b_j)_k\Big).$$

By Lemma 3.2, we obtain for $c_k = (k-1)^{k-1} + ce^{k-1}$ that $E(\sum_{0\le j<s}(b_j)^k) \le c_k\cdot n$, as claimed.

(c) is immediate from (b). □

COROLLARY 3.4. *In the situation of Lemma* 3.2 *we have*

(a) *For* $0 \le j < s$ *arbitrary*

$$\Pr(b_j \ge u) \le \begin{cases} c\cdot(e^{u-1}/u^u)\cdot(n/s)^u & \text{for } 1 \le u < k;\\ c\cdot(e^{k-1}/u^k)\cdot(n/s)^k & \text{for } k \le u.\end{cases}$$

In particular, for $s \ge n$ *and* $u \ge k$, *we have* $\Pr(b_j \ge u) = O(u^{-k})$.

(b)

$$\Pr(b_{x_0} \ge u) \le \begin{cases} c\cdot(e^{u-1}/u^u)\cdot(n/s)^u & \text{for } 1 \le u < k-1;\\ c\cdot(e^{k-2}/u^{k-1})\cdot(n/s)^{k-1} & \text{for } k-1 \le u.\end{cases}$$

In particular, for $s \ge n$ *and* $u \ge k-1$, *we have* $\Pr(b_{x_0} \ge u) = O(u^{-(k-1)})$.

(*Note.* The special case $k = u$ in Corollary 3.4(a) has already been analyzed in [15].)

Proof. (a) Assume first that $u \ge k$. Then, by Lemma 3.2(a),

$$\Pr(b_j \ge u)\cdot(u)_k = \Pr\big((b_j)_k \ge (u)_k\big)\cdot(u)_k \le E\big((b_j)_k\big) \le c\cdot\left(\frac{n}{s}\right)^k,$$

whence we get

$$\Pr(b_j \ge u) \le \frac{c}{(u)_k}\cdot\left(\frac{n}{s}\right)^k.$$

By the claim in the proof of Corollary 3.3(b), this implies

$$\Pr(b_j \ge u) \le \frac{c\cdot e^{k-1}}{u^k}\cdot\left(\frac{n}{s}\right)^k.$$

In case $1 \le u \le k-1$ it is easily seen that $\mathcal{H}$ is also (c, u)-universal. Applying the above result yields the desired estimate $\Pr(b_j \ge u) \le c\cdot e^{u-1}\cdot(n/s)^u/u^u$.

(b) The argument is exactly the same as in (a); just use Lemma 3.2(b) instead of Lemma 3.2(a). □

4. Optimal lower bounds for the deterministic case. In this and the following section we consider *deterministic* algorithms for the dictionary problem that are based on hashing and lower bounds on their performance. It will turn out that such deterministic algorithms must be much slower than the randomized algorithms described in the preceding sections.

As a basis for our lower bound proofs we introduce a simplified, abstract type of algorithm. Such algorithms maintain the following data structure D. If $S \subseteq U$ is the set of elements in the dictionary, then D consists of a rooted tree whose leaves are labeled with the elements of S. The inner nodes are labeled with hash functions whose values correspond to the edges leaving the node. In order to access a key $x \in S$, one starts at the root and repeatedly evaluates the hash function at the current node (with x as argument) to determine the edge to be followed out of the node until a leaf is reached. This leaf has label x. This data structure generalizes the one used in §2, where two hash functions had to be evaluated to access a key. We count one step for the evaluation of a hash function.

In more detail, the data structure can be described as follows. D is a rooted tree in which each inner node v is labeled with a hash function $h_v : U \to \{0, 1, \ldots, m_v - 1\}$, with $m_v \geq 2$, and has m_v children, one for each value of h_v. Each $x \in U$ determines a path from the root to a leaf. This path is given by $w_0, w_1, \ldots, w_r$, where w_0 is the root, w_{t+1} is the $h_{w_t}(x)$th child of w_t, for $0 < t < r$, and w_r is a leaf. We say that D is a dictionary for $S = \{x_0, \ldots, x_n\} \subseteq U$ if each leaf contains exactly one of the x_i. To each node v of D we associate the set $A(v) \subseteq U$ of keys that are "sent to" v by the hash functions on the path from the root to v. We define inductively: $A(v) = U$ for v the root and $A(v_q) = \{x \in A(v) \mid h_v(x) = q\}$ for $0 \leq q < m_v$ where v_q, for $0 \leq q < m_v$, are the children of v.

For our lower bound arguments, we will consider only insertions. To insert a key $x_{n+1} = x \in U$ into a dictionary D for S, we follow the path $w_0, w_1, \ldots, w_r$ determined by x, and for some node v on this path (determined by the algorithm) perform a *rehashing* at v, which means that we choose a new perfect hash function h_v for $A(v) \cap (S \cup \{x\})$. Thus, all $|A(v) \cap (S \cup \{x\})|$ children of v become leaves, and to each of them corresponds exactly one element of $A(v) \cap (S \cup \{x\})$. Such a rehashing *must* be performed for exactly one node v on the path. The cost of such an insertion is depth(v) + $|A(v) \cap (S \cup \{x\})|$. The cost of inserting $x_1, x_2, \ldots, x_n \in U$ into a dictionary D is the sum of the costs of the single insertions. Note that we assume that D initially contains one element x_0 in a leaf, with no root.

Remark 4.1. When a rehashing at v is performed, a perfect hash function for $A(v) \cap (S \cup \{x\})$ is given at linear cost; in addition, setting up the hash table, i.e, the subtree of depth 1, for this set has linear cost as well. This assumption excludes search trees that use an order on the universe U to define the way keys are distributed at nodes, as well as other schemes involving cleverly chosen hash functions that can be extended to additional keys at low cost while keeping the function injective.

Remark 4.2. (a) We require that collisions are resolved immediately by rehashing. In particular, we do not allow forming chains, i.e., linked lists, at the leaves of the tree as is done in many hashing schemes. But the absence of this restriction would not change the lower bounds by much. If we were to allow chaining, inserting n elements would cost n steps, because we could insert each element at the head of the chain, which would mean constant time per insertion. To justify our model, we have to consider tasks with insertions *and* lookups. If after inserting x we include a lookup for the element at the end of the chain into which x was inserted, then this lookup costs essentially as much as rehashing at the leaf to which the chain belongs. Thus algorithms for insertions and lookups, with chaining allowed, are at least as costly as algorithms without chaining for insertions only.

(b) One could ask if it would be advantageous to allow also rehashings at nodes v that do not lie on the path determined by the x just being inserted. But it is easily checked that the algorithm does not become slower if such rehashings are performed at the time when the last element of $S \cap A(v)$ is inserted into D. Thus it is justified not to admit such "spontaneous" rehashings.

Remark 4.3. *The role of space limitations.* In the description of the data structure D, we did not introduce the concept of the space used by D. On the other hand, some space restriction is necessary, since using the identity function as the hash function at the root would make all rehashing superfluous.

If we assume that storing a hash function h_v together with the corresponding table takes space $O(m_v) = O(|\text{range}(h_v)|)$, then $\sum_{v \text{ node in } D} m_v$ is a reasonable measure for the space used by D. In our description of the data structure D we assumed that every leaf of D contained an element of S, so for every h_v and every $j \in \{0, \dots, m_v - 1\}$ there is some $x \in S$ with $h_v(x) = j$. Since in every rooted tree with $n+1$ leaves and outdegree at least 2 the number of edges is bounded by $2n$, our data structure D satisfies $\sum_v m_v \le 2n$, which means that it needs linear space.

If the algorithm were allowed to use hash functions h_v with range larger than $|A(v) \cap S|$ when rehashing at node v, then the lower bounds given in the theorems below would still hold, with constants smaller by a factor of $\frac{1}{4}$ than those in the theorems. We only have to assume that the space used by D is not too large in relation to the size of the universe U (namely, $|U| \ge (S(n) 2 \log n)^{2 \log n} \cdot (n+1)$ in Theorem 4.4 and $|U| \ge (S(n)/k)^k$ in Theorem 4.6 for $S(n) = \sum_v m_v$). We shall comment on this in more detail below when the adversary strategies for the lower bound proofs are discussed.

We want to study the following quantities.

$T(n)$ = worst-case (amortized) cost incurred by an optimal algorithm to insert n elements.

$T_{\max}(n)$ = worst-case cost needed for a single insertion or membership query in a sequence of n instructions.

$T_k(n)$ = worst-case amortized cost needed by an optimal algorithm to insert n elements, if the depth of the tree is not allowed to exceed k, i.e., if the worst case lookup time is k.

The following three theorems sum up the results (upper and lower bounds) concerning these three quantities. Theorem 4.4 shows that amortized time $O(n)$ for n insertions cannot be achieved in the deterministic case but rather that a slowdown by a factor $\log n$ is unavoidable. Theorem 4.5 shows that in any case there will be single instructions that are very costly. If we demand constant lookup time to be guaranteed, Theorem 4.6 shows that this can only be achieved by many costly rehashings.

THEOREM 4.4. (a) $T(n) \ge (n+1) \cdot \log(n+1)$, *if* $|U| \ge (n/\log n)^{2 \log n} \cdot (n+1)$.

(b) $T(n) \le 3(n+1)\log(n+1)$.

THEOREM 4.5. (a) $T_{\max}(n) = \sqrt{n}$, *if* $|U| \ge 2(\sqrt{n})^{\sqrt{n}}$.

(b) *If only algorithms with a total cost smaller than* $f(n) \cdot n$ *for* n *insertions are considered and* $|U| \ge (n/f(n))^{2f(n)} \cdot (n+1)$, *then* $T_{\max}(n) = \Omega(n/f(n))$.

THEOREM 4.6. (a) $T_k(n) \ge (k/e) \cdot n^{1+1/k}$ *for* $n \ge e^k$, *if* $|U| > (2n/k)^k$.

(b) $T_k(n) \le d_k \cdot n^{1+1/k}$ *for all sufficiently large* n, *where the constants* d_k *can be chosen to satisfy* $d_k \sim k/e$. (*Here* $e = 2.71828 \cdots =$ *the base of the natural logarithm.*)

The *proofs* of the theorems will be given in the next section.

Remark 4.7. If we reconsider the randomized algorithm presented in §2, we see that randomization is only used for constructing perfect hash functions at expected linear cost. Thus, if we give such hash functions at guaranteed linear cost, we should obtain a deterministic algorithm that is at least not slower than the randomized one. This seems to contradict our lower bounds! To resolve this paradox, consider adversary strategies for the randomized computation model. Here the adversary has to determine the moves of the strategy without knowledge of the outcomes of the coin flips of the algorithm to be executed. This means that the data structure produced by the algorithm cannot be taken into consideration by the adversary. But this is what happens in the deterministic case and what makes the adversary as strong as indicated in the lower bounds for the deterministic model.

Remark 4.8. Theorem 4.5 gives a lower bound for our model that is bigger than the $O(\log n)$ worst-case bound for single instructions guaranteed by implementations of dictionaries as balanced search trees. This is an effect of the quite severe restriction that rehashing at a node v has cost linear in the size of the subtree rooted at v. (Cf. Remark 1.1.)

5. Proofs of the lower bounds. This section contains the proofs of the theorems stated in §4.

5.1. The adversary strategy. For proving the lower bounds, we apply an adversary argument in each case. Let us first give a general description of the adversary strategy. Initially, the tree D contains one element x_0. The adversary chooses, step by step, the element x_i to be inserted next. Basically, x_i is always chosen in such a way that it has to follow a longest path in D.

In order to be always able to find such an element x_i, we must make sure that the set of elements of U that belongs to such a longest path is not empty. The aim of the adversary is to build up long paths $w_0, w_1, w_2, \ldots$ in the tree and to make sure that the sets $A(w_0)$, $A(w_1)$, $A(w_2), \ldots$ are as large as possible. Thus, if a decision is to be made which path to choose, the adversary will, at each node v, choose that child q of v that maximizes $|h_v^{-1}[q] \cap A(v)|$, $0 \le q < m_v$. (If there is a tie, the smallest such q is chosen.) For the sake of simplicity of notation, we will assume that $q = 0$ always has this property. (If this is not the case, renumber the children of v.)

ASSUMPTION 5.1. *For all trees D ever built by the algorithms and for all nodes v of D, the set $A(v) \cap h_v^{-1}[0]$ is maximal (with respect to cardinality) among $A(v) \cap h_v^{-1}[q]$, $0 \le q < m_v$.*

We will regard the child number 0 of v as the leftmost child of v and define the leftmost path and the leftmost leaf in D accordingly (always follow the edge to child 0).

Simple adversary strategy: Choose $x_1 \neq x_0$ arbitrarily. For $i > 1$, assume that $x_1, \ldots, x_{i-1}$ have been inserted and that a tree D has been set up by the algorithm. Then let x_i be an arbitrary element of $A(v) - \{x_0, x_1, \ldots, x_{i-1}\}$, where v is the leftmost leaf of D.

Note that all elements inserted follow the leftmost path in D. This path grows as the result of inserting x_i if the algorithm chooses to perform a rehashing only at the leftmost leaf, or it is cut off at v if the algorithm performs a rehashing at an inner node v of the leftmost path.

Remark 5.2. We have made the assumption that in all nodes v of D all values of h_v are used by members of S. This has the effect that each insertion causes a collision at some node on the path to the leaf reached by the newly inserted element and, hence, causes a rehashing. Here, we wish to justify the statement made in Remark 4.3 that the lower bounds essentially also hold in the more general model where it is permitted that some values of h_v are not used by elements of S. We apply the same adversary strategy. But now it may happen that when x_i is inserted, it reaches a leaf that is not already occupied by a key from $\{x_0, \ldots, x_{i-1}\}$, hence no rehashing is necessary. However, observe that out of two subsequent insertions performed according to the adversary strategy at least one must cause a rehashing somewhere along the leftmost path. It is then seen that all lower bounds proved below hold under the assumption that not n but $2n$ keys are inserted, because they cause at least n rehashings.

The following lemma makes precise how big U has to be in order to guarantee that some suitable x_i is available in each step of the adversary strategy.

LEMMA 5.3. *Let v be a node on the leftmost path in D, and let the depth of v in D be r. Then*

(a) $|A(v)| \ge |U|/(2n/r)^r$.

(b) *If we drop the assumption (cf. Remark 4.3) that for all nodes v in the tree $A(v) \cap S \neq \emptyset$ and regard $s(D) = \sum_{v \text{ node in } D} m_v$ as a measure for the space needed by D, then for v as in* (a) *we have $|A(v)| \ge |U|/(s(D)/r)^r$.*

Proof. Let $w_0, w_1, \ldots, w_r = v$ be the path from the root w_0 to v. By definition, $|A(w_0)| = |U|$; further, $|A(w_{t+1})| \ge |A(w_t)|/m_{m_t}$, by Assumption 5.1. Thus, $|A(v)| \ge |U|/(\prod_{t=0}^{r-1} m_{w_t})$. Obviously, $\sum_{t=0}^{r-1} m_{w_t} \le s(D)$. From this it is easily seen that the denominator $\prod_{t=0}^{r-1} m_{w_t}$ cannot be larger than $(s(D)/r)^r$. This proves (b). As noted already in Remark 4.3, if $A(v) \cap S \neq \emptyset$ for all nodes v in D, then $s(D) \le 2n$. This proves (a). □

LEMMA 5.4. *Let $\bar{T}(n)$ denote the minimal number of steps needed by any algorithm for inserting n elements, if these elements are chosen according to the simple adversary strategy. (In particular, the algorithm has to admit the simple adversary strategy, which means that for each $i < n$ we have that after inserting x_i the set $A(v) - \{x_0, x_1, \ldots, x_i\}$ is nonempty, for v the leftmost leaf in D.) Then*

$$\bar{T}(n) \geq (n+1)\log(n+1).$$

Proof. (Induction on n.) Fix such an algorithm for n elements. Clearly, $\bar{T}(0) = 0$, $\bar{T}(1) = 2$ (rehashing at the root is forced). Let $n > 1$. Let $1 \leq i \leq n$ where i is maximal such that x_i is inserted by rehashing at the root. (Such an i exists, since this applies to $i = 1$.) Inserting $x_1, \ldots, x_{i-1}$ costs at least $\bar{T}(i-1)$, by the definition of $\bar{T}$; inserting x_i costs $i+1$; inserting $x_{i+1}, \ldots, x_n$ costs at least $n - i + \bar{T}(n-i)$, since the hash function at the root has to be evaluated for $x_{i+1}, \ldots, x_n$, and all these elements are sent into the leftmost subtree and have to be inserted there and are chosen according to the simple adversary strategy with respect to this subtree. (Note that this subtree already has an element.) Thus

$$\bar{T}(n) \geq \bar{T}(i-1) + (i+1) + (n-i) + \bar{T}(n-i).$$

By the induction hypothesis, this entails

$$\bar{T}(n) \geq i\log i + (n+1-i)\log(n+1-i) + n + 1,$$

and the right-hand side of the last inequality is at least $(n+1)\log(n+1)$, since the function $y\log y + (n+1-y)\log(n+1-y)$ attains its minimum in the range $1 \leq y \leq n$ in $y = (n+1)/2$. □

5.2. Proof of Theorem 4.4. We first consider the lower bound (part (a)). We would like to use the adversary strategy described above. However, to provide for the case that the leftmost path in D becomes very long and U is not as big as demanded in Lemma 5.3, we must slightly change the adversary strategy: We choose x_i so that it aims at the $\lfloor 2\log n \rfloor$th node on the leftmost path in D.

Modified adversary strategy: Choose $x_1 \neq x_0$ arbitrarily. For $i > 1$, assume that $x_1, \ldots, x_{i-1}$ have been inserted and that a tree D has been set up by the algorithm. Let $w_0, w_1, \ldots, w_r$ be the path from the root to the leftmost leaf in D. Choose x_i to be an arbitrary element of $A(w_{r'}) - \{x_0, \ldots, x_{i-1}\}$, where $r' = \min(r, \lfloor 2\log n \rfloor)$.

By Lemma 5.3, this strategy will work as long as $|U|/((2n)/(2\log n))^{2\log n} \geq n+1$, i.e., $|U| \geq (n/\log n)^{2\log n} \cdot (n+1)$.

Define

$$L = \{x_i \mid 1 \leq i \leq n, \text{depth}(v) \geq 2\log n \text{ for the vertex } v \text{ in } D \text{ at which rehashing is performed when } x_i \text{ is inserted}\}.$$

Clearly, for each $x_i \in L$ the cost of evaluating the hash functions on the way down to v alone is at least $2\log n$. We determine a lower bound for inserting the elements in $\{x_{i_1}, x_{i_2}, \ldots, x_{i_{n'}}\} = \{x_1, \ldots, x_n\} - L$ into the tree as follows. (Here, $n' = n - |L|$.) Observe that if we disregard all elements $x_i \in L$ and all inner nodes at depth $\geq 2\log n$ in the computation for $x_1, \ldots, x_n$, then we obtain a computation in which $x_{i_1}, \ldots, x_{i_{n'}}$ are inserted into a dictionary that always has depth smaller than $2\log n$ and $x_{i_1}, \ldots, x_{i_{n'}}$ are chosen according to the simple adversary strategy considered in Lemma 5.4. Thus we may conclude from Lemma 5.4 that inserting $x_{i_1}, \ldots, x_{i_{n'}}$ has cost at least $(n'+1)\log(n'+1)$. Altogether we get

$$\begin{aligned} T(n) &\geq |L| \cdot 2\log n + (n - |L| + 1) \cdot \log(n - |L| + 1) \\ &\geq \min_{0 \leq y \leq n-1} (y \cdot 2\log n + (n - y + 1) \cdot \log(n - y + 1)). \end{aligned}$$

For $n \geq 4$, the minimum is attained for $y = 0$; hence $T(n) \geq (n+1)\log(n+1)$. For $n = 1, 2, 3$, the lower bound in Theorem 4.4 is obvious. This finishes the proof of Theorem 4.4(a).

To prove the upper bound in Theorem 4.4 (part (b)), we use the following algorithm for arbitrary n: Perform a global rehashing (i. e., a rehashing at the root) for x_i if i is a power of 2. Choose the hash functions h_v, for v the root, in such a way that $|h_v^{-1}[q]| = 1$ for all $q > 0$; then all insertions that do not cause a rehashing at the root go into the leftmost subtree, to which the same algorithm is applied recursively. Let $\tilde{T}(n) =$ cost of this algorithm when applied to n elements. By inspection, $\tilde{T}(1) = 2$, $\tilde{T}(2) = 5$, $\tilde{T}(3) = 8$. We claim that $\tilde{T}(n) \leq 3(n+1)\log(n+1)$ for all n. Fix $n \geq 4$, and let $t = \lfloor \log n \rfloor$. We split $x_1, \dots, x_n$ into three groups and two single elements:

- inserting $x_1, \dots, x_{2^{t-1}-1}$ costs $\tilde{T}(2^{t-1} - 1)$;
- inserting $x_{2^{t-1}}$ costs $2^{t-1} + 1$;
- inserting $x_{2^{t-1}+1}, \dots, x_{2^t-1}$ costs $\tilde{T}(2^{t-1} - 1) + (2^{t-1} - 1)$;
- inserting x_{2^t} costs $2^t + 1$;
- inserting $x_{2^t+1}, \dots, x_n$ costs $\tilde{T}(n - 2^t) + (n - 2^t)$.

Thus, by the induction hypothesis,

$$\tilde{T}(n) \leq 2 + 2^{t-1} + n + 2 \cdot 3 \cdot 2^{t-1} \log(2^{t-1}) + 3 \cdot (n - 2^t + 1) \log(n - 2^t + 1).$$

With $2 + 2^{t-1} + n \leq 3 \cdot 2^t$ it follows that

$$\tilde{T}(n) \leq 3 \cdot 2^t \log(2^t) + 3 \cdot (n - 2^t + 1) \log(n - 2^t + 1);$$

hence, by the convexity of the function $y \log y$, we get $\tilde{T}(n) \leq 3(n+1)\log(n+1)$, as desired. This finishes the proof of Theorem 4.4(b).

5.3. Proof of Theorem 4.5. (a) Apply the simple adversary strategy from §5.1. If at some time the leftmost path in the tree becomes longer than $\sqrt{n}$, then at least one insertion had cost $\sqrt{n}$. Otherwise, the assumption $|U| \geq 2(\sqrt{n})^{\sqrt{n}}$ guarantees, by Lemma 5.3, that the adversary strategy can be carried out. Only nodes on the leftmost path have children; hence, there must be one node on the leftmost path that has at least $\sqrt{n}$ children. Thus, the cost of the last rehashing at this node was at least $\sqrt{n}$.

(b) Apply the modified adversary strategy from the proof of Theorem 4.4, for $r' = \min\{r, 2f(n)\}$. At most $n/2$ keys can be inserted below level $2f(n)$, by the overall time bound; hence, at least $n/2$ will be above that level. In levels smaller than $2f(n)$, only nodes on the leftmost path can have children; as in (a) it follows that one insertion must have had cost at least $n/2f(n)$.

5.4. Proof of Theorem 4.6.

5.4.1. The lower bound. Recall that in Theorem 4.6 only algorithms that produce trees whose depth is bounded by some constant k are considered. Let such an algorithm for inserting $x_1, \dots, x_n$ (into a table that initially contains one element x_0) be given. We use the simple adversary strategy from §5.1. From Lemma 5.3 we know that the assumption $|U| \geq (2n/k)^k$ is sufficient to ensure that $|A(v)| \geq 2$ for v the leftmost leaf of D and hence that the strategy is always applicable under this assumption.

For $k \geq 1$, $n \geq 1$ define $\hat{T}_k(n) =$ the minimal number of steps needed by any algorithm that produces only trees of depth at most k to insert $x_1, \dots, x_n$ chosen according to the adversary strategy. Clearly, $T_k(n) \geq \hat{T}_k(n)$. Trivially, $\hat{T}_k(0) = 0$ for all $k \geq 1$.

LEMMA 5.5. *$\hat{T}_k(n)$ satisfies the following inequalities.*

(a) $\hat{T}_1(n) = (n+1)(n+2)/2 - 1$, *for* $n \geq 0$.

(b) $\hat{T}_k(n) \geq \min\{l + \sum_{j=1}^{l}(ja_j + \hat{T}_{k-1}(a_j - 1)) \mid l \geq 1, a_1, \ldots, a_l \in \mathbb{N}, \sum_{j=1}^{l} a_j = n\}$, *for* $n \geq 1$, $k \geq 2$.

Proof. (a) If $k = 1$, then every element x_i is inserted by rehashing at the root, which has cost $i + 1$. Thus $\hat{T}_1(n) = \sum_{i=1}^{n}(i+1) = (n+1)(n+2)/2 - 1$.

(b) Let $x_1, \ldots, x_n$ be inserted, chosen according to the simple adversary strategy. Consider an algorithm that for inserting these elements needs $\hat{T}_k(n)$ steps. Let $x_{i_0}, x_{i_1}, \ldots, x_{i_{l-1}}$ be those elements that are inserted by global rehashing, i.e., by constructing a new perfect hash function at the root. (For x_1 this is forced, hence $i_0 = 1$.) Also, let $i_l = n + 1$. Note that between global rehashings the elements $x_{i_{j-1}+1}, \ldots, x_{i_j - 1}$ are chosen so that they are all sent to the subtree rooted at the leftmost child of the root of D and that insertions into this subtree are performed according to some strategy for $i_j - i_{j-1} - 1$ elements and depth $k - 1$; further, after the insertion of $x_{i_{j-1}}$ this subtree already has one element. By the definition of $\hat{T}_k(n)$, inserting these elements into the subtree has cost at least $\hat{T}_{k-1}(i_j - i_{j-1} - 1)$. In addition, for these elements the hash function at the root has to be evaluated, which has cost $i_j - i_{j-1} - 1$. Inserting x_{i_j}, $j = 0, 1, \ldots, l-1$, has cost $i_j + 1$. Thus the total cost is

$$\begin{aligned}\hat{T}_k(n) &\geq \sum_{j=1}^{l}\bigl((i_j - i_{j-1} - 1) + \hat{T}_{k-1}(i_j - i_{j-1} - 1)\bigr) + \sum_{j=0}^{l-1}(i_j + 1)\\ &= \sum_{j=1}^{l}\hat{T}_{k-1}(i_j - i_{j-1} - 1) + \sum_{j=1}^{l} i_j.\end{aligned}$$

Let $a_{l+1-j} = i_j - i_{j-1}$ for $1 \leq j \leq l$. Then $\sum_{j=1}^{l} a_j = i_l - i_0 = n$ and $\sum_{j=1}^{l} i_j = l \cdot i_0 + \sum_{j=1}^{l}(l + 1 - j) \cdot a_{l+1-j}$; hence,

$$\hat{T}_k(n) \geq \sum_{j=1}^{l}\bigl(j \cdot a_j + \hat{T}_{k-1}(a_j - 1)\bigr) + l.$$

This proves part (b). □

The proof of Theorem 4.6 is completed by the following lemma.

LEMMA 5.6.

$$\hat{T}_k(n) \geq g_k(n+1)$$

for all $k \geq 1$, $n \geq 0$, *where*

$$g_k(y) = \begin{cases} 0 & \textit{for} \quad y = 0;\\ y \ln y & \textit{for} \quad 0 < y \leq e^k;\\ \dfrac{k}{e} \cdot y^{1+1/k} & \textit{for} \quad e^k < y.\end{cases}$$

For the *proof* of this lemma see the appendix. It is a technical argument based solely on the inequalities of Lemma 5.5.

5.4.2. The upper bound. We will describe an algorithm for inserting n elements $x_1, \ldots, x_n \in U$ into a table (which initially contains one element x_0) so that the depth of the resulting tree never becomes larger than k. As in the proof of Theorem 4.4(b), the hash function h_v chosen for a vertex v always satisfies $|h_v^{-1}[q] \cap A(v)| = 1$ for all $q > 0$. This means that subsequent elements that are inserted in the subtree rooted at v are always sent to the leftmost subtree of v. Let

$$d_1 = 1, \qquad d_k = k \cdot \frac{k+1}{k+2} \cdot \left(\frac{d_{k-1}}{k-1}\right)^{(k-1)/k} \qquad \text{for } k > 1.$$

Then $d_k = k \cdot \left(\prod_{q=2}^{k}\big((q+1)/(q+2)\big)^q\right)^{1/k}$. As an abbreviation, let $b_k = d_k/k$.

Algorithm for a table of depth at most k (inductive description):
$k = 1$: Insert each element by global rehashing.
$k > 1$: Let $i_t = \left\lceil \sum_{s=1}^{t}\big(s/(k \cdot b_{k-1})\big)^{k-1} \right\rceil$, for $t = 0, 1, 2, 3, \dots$. Insert the elements $x_1 = x_{i_1}, x_{i_2}, x_{i_3}, \dots$ by global rehashing; that is, by establishing a new hash function h_v at the root v. Between these global rehashings the elements $x_{i_{t-1}+1}, \dots, x_{i_t-1}$ all go into the leftmost subtree of the root. Apply the algorithm for depth at most $k-1$ to this subtree, for these $i_t - i_{t-1} - 1$ elements.

It is obvious that this algorithm always maintains a tree of depth at most k; hence, a lookup time of k is guaranteed. We only have to analyze the time required for insertions. For $k \geq 1$, $n \geq 0$, let

$$\tilde{T}_k(n) = \text{cost of inserting } x_1, \dots, x_n \text{ into a table, which initially has one element, using the algorithm just described.}$$

(Note that for the cost of the algorithm it is irrelevant which particular elements $x_1, \dots, x_n$ are inserted.) To finish the proof of Theorem 4.6(b), we just have to show the following.

LEMMA 5.7. (a) $\tilde{T}_k(n) \leq d_k \cdot n^{1+1/k}$ *for all* $n \geq n_k$, *for* n_k *large enough (for all* $k \geq 1$).
(b) $\lim_{k\to\infty} d_k/(k/e) = 1$.

Proof. (a) (Induction on k.)
Initial step ($k = 1$): Obviously, $\tilde{T}_1(n) = (n+1)(n+2)/2 - 1 \leq n^2$ for $n \geq 3$.

Induction step ($k > 1$): Assume $\tilde{T}_{k-1} \leq d_{k-1} \cdot n^{k/(k-1)}$ for all $n \geq n_{k-1}$. Now let n be fixed, n large enough. Define $t_0 = \min\{t \geq 1 \mid i_t > n\}$, for the sequence i_t, $t \geq 1$, defined in the strategy. We first estimate t_0. Clearly, by the definition of i_t and t_0 we have

$$\sum_{s=1}^{t_0-1} s^{k-1} \leq (k \cdot b_{k-1})^{k-1} \cdot n < \sum_{s=1}^{t_0} s^{k-1},$$

hence (by estimating the sums by integrals and taking kth roots),

$$t_0 - 1 \leq k \cdot b_{k-1}^{(k-1)/k} \cdot n^{1/k} < t_0 + 1\,. \tag{5.1}$$

In the following, we estimate $\tilde{T}_k(i_{t_0} - 1)$, which certainly is an upper bound for $\tilde{T}_k(n)$. We let $i_0 = 0$. Then inserting the element x_{i_t} (by global rehashing) has cost i_t+1, for $t = 1, 2, \dots, t_0 - 1$; inserting the elements $x_{i_{t-1}+1}, \dots, x_{i_t-1}$ has cost $(i_t - i_{t-1} - 1) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)$, for $t = 1, 2, \dots, t_0$. Thus,

$$\tilde{T}_k(n) \leq \sum_{t=1}^{t_0}\big((i_t - i_{t-1} - 1) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)\big) + \sum_{t=1}^{t_0-1}(i_t + 1),$$

or, after a trivial transformation,

$$\tilde{T}_k(n) \leq \sum_{t=1}^{t_0}\big((t_0 + 1 - t)(i_t - i_{t-1}) + \tilde{T}_{k-1}(i_t - i_{t-1} - 1)\big).$$

Substituting the induction hypothesis $\tilde{T}_{k-1}(n') \leq d_{k-1} \cdot (n')^{k/(k-1)}$, for $n' \geq n_{k-1}$, into this inequality yields

$$\tilde{T}_k(n) \leq \sum_{t=1}^{t_0}\big((t_0 + 1 - t)(i_t - i_{t-1} - 1) + d_{k-1}(i_t - i_{t-1} - 1)^{k/(k-1)}\big) + \tfrac{1}{2}t_0(t_0 + 1) + E_k,$$

for some constant E_k (needed to make up for the error caused by replacing $\tilde{T}_{k-1}(i_t - i_{t-1} - 1)$ by $d_{k-1}(i_t - i_{t-1} - 1)^{k/(k-1)}$ for t so small that $i_t - i_{t-1} - 1 < n_{k-1}$). By the definition of i_t we clearly have $i_t - i_{t-1} - 1 \le \big(t/(kb_{k-1})\big)^{k-1}$; furthermore, from the bounds on t_0 in (5.1) it follows that $t_0^2 = O\big((n^{1/k})^2\big) = O(n)$. Thus,

$$\begin{aligned}\tilde{T}_k(n) &\le \sum_{t=1}^{t_0}\left[\left(\frac{t}{kb_{k-1}}\right)^{k-1} \cdot (t_0 + 1 - t) + d_{k-1} \cdot \left(\frac{t}{kb_{k-1}}\right)^{k}\right] + O(n)\\ &= (kb_{k-1})^{1-k} \cdot \left(\sum_{t=1}^{t_0}(t_0+1)t^{k-1} - \sum_{t=1}^{t_0} t^k/k\right) + O(n).\end{aligned}$$

We substitute the two inequalities $\sum_{t=1}^{t_0} t^{k-1} \le (t_0 + 1)^k/k$ and $\sum_{t=1}^{t_0} t^k \ge t_0^{k+1}/(k+1)$ (obtained by replacing the sums by integrals) and simplify, noting that $(t_0 + 1)^{k+1} = t_0^{k+1} + O(t_0^k)$. In this way we get

$$\tilde{T}_k(n) \le \frac{1}{k+1} \cdot (kb_{k-1})^{1-k} \cdot t_0^{k+1} + O(t_0^k) + O(n).$$

By (5.1), we have $t_0^k = O\big((n^{1/k})^k\big) = O(n)$ and furthermore that $t_0^{k+1} = k^{k+1} \cdot b_{k-1}^{k-1/k} \cdot n^{1+1/k} + O(t_0^k)$. Hence

$$\begin{aligned}\tilde{T}_k(n) &\le \frac{1}{k+1} \cdot (kb_{k-1})^{1-k} \cdot k^{k+1} \cdot b_{k-1}^{k-1/k} \cdot n^{1+1/k} + O(n)\\ &= \frac{k^2}{k+1} \cdot b_{k-1}^{(k-1)/k} \cdot n^{1+1/k} + O(n).\end{aligned}$$

For n large enough, this implies

$$\tilde{T}_k(n) \le k \cdot \frac{k+1}{k+2} \cdot b_{k-1}^{(k-1)/k} \cdot n^{1+1/k} = d_k \cdot n^{1+1/k},$$

and this is what we wanted to show.

(b) By definition, $d_k/k = \left(\prod_{q=2}^{k}((q+1)/(q+2))^q\right)^{1/k}$. Recall that

$$\left(\frac{q+1}{q+2}\right)^{q+2} \le \frac{1}{e} \le \left(\frac{q+1}{q+2}\right)^{q+1}$$

for all q, and hence

$$\frac{d_k}{k} \cdot \left(\prod_{q=2}^{k}\left(\frac{q+1}{q+2}\right)^2\right)^{1/k} \le \left(\frac{1}{e}\right)^{(k-1)/k} \le \frac{d_k}{k} \cdot \left(\prod_{q=2}^{k}\left(\frac{q+1}{q+2}\right)\right)^{1/k}.$$

Clearly,

$$\lim_{k\to\infty}\left(\prod_{q=2}^{k}\frac{q+1}{q+2}\right)^{1/k} = \lim_{k\to\infty}\left(\frac{3}{k+1}\right)^{1/k} = 1,$$

and thus $\lim_{k\to\infty} d_k/k = 1/e$, as claimed. □

A. Appendix.

A.1. Proof of Lemma 5.6. We show the following: If the functions $T_k, k \geq 1$, satisfy slightly weaker inequalities than those stated in Lemma 5.5, namely, $T_k(0) = 0$ for all $k \geq 1$, and

(a) $T_1(n) \geq (n+1)(n+2)/2 - 1$ for all $n \geq 1$,

(b) $T_k(n) \geq \min\{\, 1 + \sum_{j=1}^{l}(ja_j + T_{k-1}(a_j - 1)) \mid l \geq 1, a_1, \ldots, a_l \in \mathbb{N}, \sum_{j=1}^{l} a_j = n\,\}$, for all $n \geq 1, k \geq 2$, then the functions T_k satisfy the assertion of Lemma 5.6, that is,

$$T_k(n) \geq g_k(n+1)$$

for all $k \geq 1, n \geq 0$, where, for $k \geq 1$,

$$g_k(y) = \begin{cases} 0 & \text{if } y = 0; \\ y \ln y & \text{if } 0 < y \leq e^k; \\ (k/e) \cdot y^{1+1/k} & \text{if } e^k < y. \end{cases}$$

We proceed by induction on k. For $k = 1$, it is easily checked that $g_1(n+1) \leq (n+1)(n+2)/2 - 1$ for all $n \geq 0$. Thus, let $k > 1$, and assume the claim to be true for $k-1$; that is, $T_{k-1}(n) \geq g_{k-1}(n+1) = g(n+1)$, for all $n \geq 0$. (From here on, we will write g for g_{k-1}.) For $n = 0$, the claim is trivially satisfied. Let $n \geq 1$ be fixed. By assumption (b) above and the induction hypothesis, we may fix some $l \geq 1$ and a sequence $a = (a_1, \ldots, a_l)$ of natural numbers with $\sum_{j=1}^{l} a_j = n$ and

$$T_k(n) - 1 \geq \sum_{j=1}^{l}(ja_j + T_{k-1}(a_j - 1)) \geq \sum_{j=1}^{l}(ja_j + g(a_j)). \tag{A.1}$$

We want to find a lower bound on the last sum in (A.1). The first step we take is to transform sums to integrals and sequences of natural numbers to real functions. The sequence a may be regarded as equivalent to the piecewise constant function $f_a \colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$ defined by

$$f_a(x) = \begin{cases} a_j & \text{if } j - 1 \leq x < j,\ j = 1, \ldots, l; \\ 0 & \text{if } l \leq x < \infty. \end{cases}$$

The condition $\sum_{j=1}^{l} a_j = n$ translates to $\int_0^\infty f_a(x)\,dx = n$, and the sum in (A.1) can be expressed as

$$\sum_{j=1}^{l}(ja_j + g(a_j)) = \int_0^\infty \Big(xf_a(x) + g(f_a(x))\Big)\,dx + \frac{n}{2}. \tag{A.2}$$

Our aim is now to find a lower bound on the integral in (A.2). To this end, we transform the minimization problem a little further: instead of piecewise constant functions such as f_a we will consider continuous functions.

DEFINITION A.1. (a) *Let $\mathcal{D}$ be the class of all continuous functions $f \colon \mathbb{R}_0^+ \to \mathbb{R}^+$ (strictly positive) so that $\int_0^\infty f(x)\,dx = n$ and so that* $\lim_{x\to\infty} e^x f(x)$ *exists and is positive.*

(b) *Let $G \colon \mathbb{R}_0^+ \times \mathbb{R}_0^+ \to \mathbb{R}$ be defined by $G(x, y) = xy + g(y) = xy + g_{k-1}(y)$.*

(c) *For $f \in \mathcal{D}$ let $I(f) = \int_0^\infty G(x, f(x))\,dx$. (Note that the condition* $\lim_{x\to\infty} e^x f(x) > 0$ *ensures that the integral exists.)*

It is easy to see that for any given $\varepsilon > 0$ the piecewise constant function f_a can be approximated by some $f_{a,\varepsilon} \in \mathcal{D}$ in such a way that

$$I(f_{a,\varepsilon}) < \varepsilon + \int_0^\infty xf_a(x) + g(f_a(x))\,dx. \tag{A.3}$$

Now it follows from (A.1), (A.2), (A.3), and the fact that $f_{a,\varepsilon} \in \mathcal{D}$ for all $\varepsilon > 0$ that

$$T_k(n) - \left(\frac{n}{2} + 1\right) \geq \inf\{ I(f) \mid f \in \mathcal{D}\}. \tag{A.4}$$

The following proposition establishes the existence of a function $f_0 \in \mathcal{D}$ that realizes this infimum; moreover, it provides an equation for f_0 that will enable us to calculate f_0 explicitly. Then we may evaluate $I(f_0)$ to obtain the desired lower bound on $T_k(n)$. The proposition is proved by reducing the problem of minimizing $I(f)$ over $\mathcal{D}$ to a standard situation treated in the Calculus of Variations. (The details of this proof, which will be given in the second part of the appendix, are irrelevant for the rest of the argument.)

PROPOSITION A.2. *There is a unique function $f_0 \in \mathcal{D}$ such that*

$$I(f_0) = \min\{ I(f) \mid f \in \mathcal{D}\}. \tag{A.5}$$

Moreover, there is some constant $A \in \mathbb{R}$ such that f_0 satisfies

$$\left.\frac{\partial}{\partial y} G(x, y)\right|_{y=f_0(x)} = A \quad \textit{for all } x \in \mathbb{R}_0^+. \tag{A.6}$$

Our next goal is to use (A.6) in order to obtain an expression for f_0. First, we calculate A. By the definition of G, we have that $(\partial/\partial y)G(x, y) = x + g'(y)$, and hence (A.6) becomes

$$x + g'(f_0(x)) = A \quad \text{for } x \geq 0. \tag{A.7}$$

It follows easily from the definition of $g = g_{k-1}$ that

$$g'(y) = \begin{cases} 1 + \ln y & \text{if } 0 < y \leq e^{k-1}; \\ (k/e) \cdot y^{1/(k-1)} & \text{if } e^{k-1} \leq y < \infty. \end{cases}$$

Obviously, $g'(y)$ is a strictly increasing function of y with range $\mathbb{R}$, and the inverse of g' is given by

$$(g')^{-1}(z) = \begin{cases} e^{z-1} & \text{if } -\infty < z \leq k; \\ (ez/k)^{k-1} & \text{if } k \leq z < \infty. \end{cases} \tag{A.8}$$

Thus (A.7) can be transformed to

$$f_0(x) = (g')^{-1}(A - x) \quad \text{for } x \geq 0. \tag{A.9}$$

Since $f_0 \in \mathcal{D}$, we have (using the explicit formula (A.8) for $(g')^{-1}$):

$$n = \int_0^\infty (g')^{-1}(A - x)\, dx = \begin{cases} e^{A-1} & \text{if } A \leq k; \\ (e^{k-1}/k^k) \cdot A^k & \text{if } A \geq k. \end{cases} \tag{A.10}$$

We may now solve (A.10) for A to obtain

$$A = \begin{cases} 1 + \ln n & \text{if } n \leq e^{k-1}; \\ (n/e^{k-1})^{1/k} \cdot k & \text{if } n \geq e^{k-1}. \end{cases} \tag{A.11}$$

Now, finally, we are in a position to evaluate $I(f_0)$. First, we substitute (A.9) into the definition of $I(f_0)$ (see Definition A.1(b),(c)) to obtain

$$I(f_0) = \int_0^\infty x \cdot (g')^{-1}(A - x) + g((g')^{-1}(A - x))\, dx. \tag{A.12}$$

Case 1: $n \geq e^{k-1}$. Then $A - k \geq 0$, and we get from (A.12), by substituting (A.8) and the definition of $g = g_{k-1}$, that

$$I(f_0) = \int_0^{A-k} x \cdot (e(A-x)/k)^{k-1} + \frac{k-1}{e} \cdot (e(A-x)/k)^k \, dx$$

$$+ \int_{A-k}^{\infty} x \cdot e^{A-x-1} + e^{A-x-1} \cdot \ln(e^{A-x-1}) \, dx.$$

The second integral evaluates to $(A-1)e^{k-1}$; the first one equals

$$A \cdot \left(\frac{e}{k}\right)^{k-1} \cdot \int_0^{A-k} (A-x)^{k-1} \, dx - \frac{e^{k-1}}{k^k} \cdot \int_0^{A-k} (A-x)^k \, dx$$

$$= A \cdot e^{k-1} \cdot \frac{1}{k^k} \cdot (A^k - k^k) - \frac{e^{k-1}}{k^k} \cdot \frac{1}{k+1}(A^{k+1} - k^{k+1})$$

$$= A^{k+1} \cdot e^{k-1} \cdot \frac{1}{k^k} \cdot \frac{k}{k+1} - A \cdot e^{k-1} + e^{k-1} \cdot \frac{k}{k+1}.$$

Altogether,

$$I(f_0) = A^{k+1} e^{k-1} \cdot \frac{1}{k^k} \cdot \frac{k}{k+1} - \frac{1}{k+1} \cdot e^{k-1}.$$

Substituting the value $A = (n/e^{k-1})^{1/k} \cdot k$ given by (A.11) into the last equation and using the fact that $k/(k+1) \geq e^{-1/k}$ results in

$$I(f_0) = \frac{k}{k+1} \cdot e^{-1+1/k} \cdot k \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1} \geq \frac{k}{e} \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1}.$$

In combination with (A.4) and (A.5) this yields

$$T_k(n) \geq \frac{k}{e} \cdot n^{1+1/k} - \frac{e^{k-1}}{k+1} + \frac{n}{2} + 1. \tag{A.13}$$

Elementary estimates (which use the assumption that $n \geq e^{k-1}$) show that the right-hand side of (A.13) is bounded from below by $(n+1)\ln(n+1)$ if $n+1 \leq e^k$ and by $(k/e) \cdot (n+1)^{1+1/k}$ if $n+1 \geq e^k$. This proves the inequality $T_k(n) \geq g_k(n+1)$ in Case 1.

Case 2: $n \leq e^{k-1}$. Then $A - k \leq 0$, and we get from (A.12) and (A.8) that

$$I(f_0) = \int_0^{\infty} x \cdot e^{A-x-1} + e^{A-x-1} \cdot \ln(e^{A-x-1}) \, dx = (A-1) \cdot e^{A-1}.$$

We now substitute the value $A = 1 + \ln n$ from (A.11) to obtain that $I(f_0) = n \ln n$. In combination with (A.5) and (A.4) this entails that

$$T_k(n) \geq n \ln n + \frac{n}{2} + 1.$$

Elementary estimates show that the right-hand side of this inequality is bounded from below by $(n+1)\ln(n+1) = g_k(n+1)$. This finishes the proof of Lemma 5.6.

A.2. Proof of Proposition A.2. We sketch a proof of Proposition A.2 stated in the first part of this appendix. We reduce the proposition to a standard theorem from the Calculus of Variations. First, instead of dealing with conditions defined by the *integrals* of the functions in class $\mathcal{D}$ (see Definition A.1(a)) we need conditions on the *values* of the functions considered at the boundaries of the interval. For this, we consider the integral functions $x \mapsto \int_0^x f(\xi)\,d\xi$, for $f \in \mathcal{D}$, $x \in \mathbb{R}_0^+$. Second, we transform the unbounded interval $\mathbb{R}_0^+$ to the bounded interval $[0, 1]$ by means of the transformation $x = x(t) = -\ln(1-t)$, for $0 \le t < 1$, with inverse transformation $t = t(x) = 1 - e^{-x}$, for $0 \le x < \infty$.

DEFINITION A.3. (a) *Let $\mathcal{E}$ be the class of all functions $\varphi\colon [0,1] \to \mathbb{R}_0^+$ that have a continuous derivative $(d/dt)\varphi(t) = \varphi'(t) > 0$ in $[0,1]$ and satisfy $\varphi(0) = 0$ and $\varphi(1) = n$.*

(b) *Let $H\colon [0,1] \times \mathbb{R}_0^+ \to \mathbb{R}$ be defined by*

$$H(t,z) = \begin{cases} \frac{1}{1-t} \cdot G(-\ln(1-t), (1-t)z) & \text{if } z > 0 \text{ and } 0 \le t < 1, \\ z \ln z & \text{if } z > 0 \text{ and } t = 1, \end{cases}$$

where $G(x,y) = xy + g(y)$ is as in Definition A.1(b).

(c) *For $\varphi \in \mathcal{E}$ let $J(\varphi) = \int_0^1 H(t, \varphi'(t))\,dt$. (The following lemma implies that the integral is well defined.)*

LEMMA A.4. *The function H from the previous definition is continuous, and for each fixed $t \in [0,1]$ the function $z \mapsto (\partial^2/\partial z^2)H(t,z)$, $z \in \mathbb{R}$, is continuous and strictly positive, excepting for $t \neq 1$ and $y = e^{k-1}/(1-t)$.*

Proof. Straightforward verification. □

LEMMA A.5. *There is a bijection between $\mathcal{D}$ and $\mathcal{E}$ given by the mappings $f \mapsto \varphi_f$ and $\varphi \mapsto f_\varphi$, where*

$$\varphi_f(t) = \begin{cases} \displaystyle\int_0^{-\ln(1-t)} f(\xi)\,d\xi & \text{if } 0 \le t < 1, \\ n & \text{if } t = 1, \end{cases}$$

and

$$f_\varphi(x) = \tfrac{d}{dx}\varphi(1 - e^{-x}) = \varphi'(1 - e^{-x}) \cdot e^{-x} \quad \text{if } \; 0 \le x \le \infty.$$

Moreover, we have $I(f) = J(\varphi_f)$, for all $f \in \mathcal{D}$.

Proof. Straightforward verification. □

We now need the following theorem, which is obtained by combining Proposition (3.10) and Theorem (3.7) from [18].

THEOREM A.6. *If $H = H(t,z)$ is continuous on $[0,1] \times \mathbb{R}_0^+$ and if, for each $t \in [0,1]$, the function $z \mapsto (\partial^2/\partial z^2)H(t,z)$ is continuous and positive (except possibly at a finite set of z-values), then there is exactly one function $\varphi_0 \in \mathcal{E}$ that minimizes $\int_0^1 H(t, \varphi'(t))\,dt$ on $\mathcal{E}$. Moreover, this function φ_0 satisfies $(\partial/\partial z)H(t,z)|_{z=\varphi_0'(t)} = \text{const}$, for $t \in [0,1]$.*

By Lemma A.4, the function H from Definition A.3 satisfies the hypothesis of this theorem, and hence there is a unique function $\varphi_0 \in \mathcal{E}$ that minimizes $J(\varphi)$ over $\mathcal{E}$; moreover, there is some $A \in \mathbb{R}$ with $(\partial/\partial z)H(t,z)|_{z=\varphi_0'(t)} = A$ for all $t \in [0,1]$. By Lemma A.5, the function $f_0 = f_{\varphi_0}$ minimizes $I(f)$ over $\mathcal{D}$. It remains to establish equation (A.6).

By Definition A.3(b) we have $(\partial/\partial z)H(t,z) = (\partial/\partial y)G(-\ln(1-t), y)|_{y=(1-t)z}$, and hence

$$\text{(A.14)} \qquad A = \frac{\partial}{\partial z}H(t,z)\bigg|_{z=\varphi_0'(t)} = \frac{\partial}{\partial y}G(-\ln(1-t), y)\bigg|_{y=(1-t)\varphi_0'(t)} \qquad \text{for } 0 \le t \le 1.$$

By Lemma A.5, we have $f_0(x) = (d/dx)\varphi_0(1 - e^{-x}) = \varphi_0'(1 - e^{-x}) \cdot e^{-x} = \varphi_0'(t) \cdot (1 - t)$, under the bijection $t \mapsto x(t) = -\ln(1 - t)$. Hence, equation (A.14) entails that $(\partial/\partial y)G(x, y)|_{y=f_0(x)} = A$ for all $x \in \mathbb{R}_0^+$, as claimed. This finishes the proof of Proposition A.2.

REFERENCES

[1] H. V. Aho and D. Lee, *Storing a dynamic sparse table*, Proc. 27th IEEE Symp. on Foundations of Computer Science, 1986, pp. 55–60.

[2] H. Bast and T. Hagerup, *Fast and reliable parallel hashing*, Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 50–61.

[3] G. Brassard and S. Kannan, *The generation of random permutations on the fly*, Inform. Process. Lett., 28 (1988), pp. 207–212.

[4] J. L. Carter and M. N. Wegman, *Universal classes of hash functiôns*, J. Comput. Syst. Sci., 18 (1979), pp. 143–154.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, *Dynamic perfect hashing: Upper and lower bounds*, Proc. 29th IEEE Symp. on Foundations of Computer Science, 1988, pp. 524–531.

[6] M. Dietzfelbinger and F. Meyer auf der Heide, *How to distribute a dictionary in a complete network*, Proc. 22nd Annual ACM Symp. on Theory of Computing, 1990, pp. 117–127.

[7] ———, *A new universal class of hash functions and dynamic hashing in real time*, Proc. 17th ICALP, Lecture Notes in Comput. Sci., 443, Springer-Verlag, New York, 1990, pp. 6–19. Final version in J. Buchmann, H. Ganzinger, and W. J. Paul, eds., Informatik · Festschrift zum 60. Geburtstag von Günter Hotz, B. G. Teubner, Stuttgart, 1992, pp. 95–119.

[8] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, *A reliable randomized algorithm for the closest-pair problem*, Forschungsbericht Nr. 513, Fachbereich Informatik, Universität Dortmund, 1993.

[9] M. L. Fredman, J. Komlós, and E. Szemerédi, *Storing a sparse table with $O(1)$ worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538–544.

[10] J. Gil and Y. Matias, *Fast hashing on a PRAM — Designing by expectation*, Proc. 2nd Annual ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 271–280.

[11] G. H. Gonnet, *Expected length of the longest probe sequence in hash code searching*, J. Assoc. Comput. Mach., 28 (1981), pp. 289–304.

[12] D. E. Knuth , *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[13] K. Mehlhorn , *Data Structures and Algorithms, Vol. 1: Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[14] K. Mehlhorn, S. Näher, and M. Rauch, *On the complexity of a game related to the dictionary problem*, SIAM J. Comput., 19 (1990), pp. 902–906.

[15] K. Mehlhorn and U. Vishkin, *Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory*, Acta Inform., 21 (1984), pp. 339–374.

[16] A. Siegel, *On universal classes of fast hash functions, their time-space tradeoff, and their applications*, Proc. 30th IEEE Symp. on Foundations of Computer Science, 1989, pp. 20–25.

[17] R. Sundar, *A lower bound for the dictionary problem under a hashing model*, Proc. 32nd IEEE Symp. on Foundations of Computer Science, 1991, pp. 612–621.

[18] J. L. Troutman, *Variational calculus with elementary convexity*, Springer-Verlag, New York, 1983.

[19] M. N. Wegman and J. L. Carter, *New classes and applications of hash functions*, Proc. 20th IEEE Symp. on Foundations of Computer Science, 1979, pp. 175–182.

[20] M. Wenzel, *Eine Implementierung von Dynamic Perfect Hashing*, Diplomarbeit, Universität des Saarlandes, 1990.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 762–779, August 1994

MEASURE, STOCHASTICITY, AND THE DENSITY OF HARD LANGUAGES*

JACK H. LUTZ[†] AND ELVIRA MAYORDOMO[‡]

Abstract. The main theorem of this paper is that, for every real number $\alpha < 1$ (e.g., $\alpha = 0.99$), only a measure 0 subset of the languages decidable in exponential time are $\leq^{\mathrm{P}}_{n^\alpha-tt}$-reducible to languages that are not exponentially dense. *Thus every* $\leq^{\mathrm{P}}_{n^\alpha-tt}$*-hard language for* E *is exponentially dense.* This strengthens Watanabe's 1987 result, that every $\leq^{\mathrm{P}}_{O(\log n)-tt}$-hard language for E is exponentially dense. The combinatorial technique used here, the *sequentially most frequent query selection*, also gives a new, simpler proof of Watanabe's result.

The main theorem also has implications for the structure of NP under strong hypotheses. Ogiwara and Watanabe (1991) have shown that the hypothesis $\mathrm{P} \neq \mathrm{NP}$ implies that every $\leq^{\mathrm{P}}_{btt}$-hard language for NP is nonsparse (i.e., not polynomially sparse). Their technique does not appear to allow significant relaxation of either the query bound or the sparseness criterion. It is shown here that a stronger hypothesis—namely, that NP does not have measure 0 in exponential time—implies the stronger conclusion that, for every real $\alpha < 1$, every $\leq^{\mathrm{P}}_{n^\alpha-tt}$-hard language for NP is exponentially dense. Evidence is presented that this stronger hypothesis is reasonable.

The proof of the main theorem uses a new, very general *weak stochasticity theorem*, ensuring that almost every language in E is statistically unpredictable by feasible deterministic algorithms, even with linear nonuniform advice.

Key words. complexity classes, computational complexity, dense languages, polynomial reductions, resource-bounded measure, sparse languages, weak stochasticity

AMS subject classification. 68Q15

1. Introduction.

How dense must a language $A \subseteq \{0, 1\}^*$ be in order to be hard for a complexity class $\mathcal{C}$? The ongoing investigation of this question, especially important when $\mathcal{C} = \mathrm{NP}$, has yielded several significant results [3], [12], [20], [22], [23], [30], [31] over the past fifteen years.

Any formalization of this question must specify the class $\mathcal{C}$ and give precise meanings to "hard" and "how dense." The results of this paper concern the classes $\mathrm{E} = \mathrm{DTIME}(2^{\mathrm{linear}})$, $\mathrm{E}_2 = \mathrm{DTIME}(2^{\mathrm{polynomial}})$, and all subclasses $\mathcal{C}$ of these classes, though we are particularly interested in the case $\mathcal{C} = \mathrm{NP}$.

We will consider the polynomial-time reducibilities $\leq^{\mathrm{P}}_m$ (*many-one* reducibility), $\leq^{\mathrm{P}}_{\mathrm{T}}$ (*Turing* reducibility), $\leq^{\mathrm{P}}_{btt}$ (*bounded truth-table* reducibility), and $\leq^{\mathrm{P}}_{q(n)-tt}$ (truth-table reducibility with $q(n)$ queries on inputs of length n, where $q : \mathbf{N} \to \mathbf{Z}^+$). If $\leq^{\mathrm{P}}_r$ is any of these reducibilities, we say that a language A is $\leq^{\mathrm{P}}_r$*-hard* for a class $\mathcal{C}$ of languages if $\mathcal{C} \subseteq \mathrm{P}_r(A)$, where $\mathrm{P}_r(A) = \{\, B \subseteq \{0, 1\}^* \mid B \leq^{\mathrm{P}}_r A\}$.

Two criteria for "how dense" a language A is have been widely used. A language A is (*polynomially*) *sparse*, and we write $A \in \mathrm{SPARSE}$, if there is a polynomial p such that $|A_{\leq n}| \leq p(n)$ for all $n \in \mathbf{N}$, where $A_{\leq n} = A \cap \{0, 1\}^{\leq n}$. A language A is (*exponentially*) *dense*, and we write $A \in \mathrm{DENSE}$, if there is a real number $\epsilon > 0$ such that $|A_{\leq n}| \geq 2^{n^\epsilon}$ for all sufficiently large $n \in \mathbf{N}$. It is clear that no sparse language is dense.

For any of the above choices of the reducibility $\leq^{\mathrm{P}}_r$, *all known* $\leq^{\mathrm{P}}_r$*-hard languages for* NP *are dense*. Efforts to explain this observation (and similar observations for other classes and reducibilities) have yielded many results. (See [9] for a thorough survey.) We mention four such results that are particularly relevant to the work presented here.

*Received by the editors September 24, 1992; accepted for publication (in revised form) April 23, 1993.

[†] Department of Computer Science, Iowa State University, Ames, Iowa 50011 (lutz@iastate.edu). This author's research was supported in part by National Science Foundation grant CCR-9157382, with matching funds from Rockwell International, and by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), where he was a visitor during the first phase of this work.

[‡] Department of Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028 Barcelona, Spain. This author's research, performed while visiting Iowa State University, was supported in part by Spanish Government grant FPI PN90 and by National Science Foundation grant CCR-9157382.

Let DENSE^c denote the complement of DENSE, i.e., the set of all languages A such that, for all $\epsilon > 0$, there exist infinitely many n such that $|A_{\leq n}| < 2^{n^\epsilon}$. For each reducibility $\leq_r^{\mathrm{P}}$ and set $\mathcal{S}$ of languages, we write

$$\mathrm{P}_r(\mathcal{S}) = \bigcup_{A \in \mathcal{S}} \mathrm{P}_r(A).$$

The first result on the density of hard languages was the following.

THEOREM 1 (Meyer [22]). *Every $\leq_m^{\mathrm{P}}$-hard language for* E (*or any larger class*) *is dense. That is,*

$$\mathrm{E} \not\subseteq \mathrm{P}_m(\mathrm{DENSE}^c).$$

Theorem 1 was subsequently improved to truth-table reducibility with $O(\log n)$ queries in the next theorem.

THEOREM 2 (Watanabe [31], [30]). *Every $\leq_{O(\log n)-tt}^{\mathrm{P}}$-hard language for* E *is dense. That is,*

$$\mathrm{E} \not\subseteq \mathrm{P}_{O(\log n)-tt}(\mathrm{DENSE}^c).$$

Regarding NP, Berman and Hartmanis [3] conjectured that no sparse language is $\leq_m^{\mathrm{P}}$-hard for NP, unless P = NP. This conjecture was subsequently proven correct as seen in the following.

THEOREM 3 (Mahaney [20]). *If* P $\neq$ NP, *then no sparse language is $\leq_m^{\mathrm{P}}$-hard for* NP. *That is,*

$$\mathrm{P} \neq \mathrm{NP} \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_m(\mathrm{SPARSE}).$$

Theorem 3 has recently been extended to truth-table reducibility with a bounded number of queries.

THEOREM 4 (Ogiwara and Watanabe [23]). *If* P $\neq$ NP, *then no sparse language is $\leq_{btt}^{\mathrm{P}}$-hard for* NP. *That is,*

$$\mathrm{P} \neq \mathrm{NP} \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{btt}(\mathrm{SPARSE}).$$

The Main Theorem of this paper, Theorem 14, extends Theorems 1 and 2 by showing that, for every real $\alpha < 1$ (e.g., $\alpha = 0.99$), only a measure 0 subset of the languages in E are $\leq_{n^\alpha-tt}^{\mathrm{P}}$-reducible to nondense languages. "Measure 0 subset" here refers to the resource-bounded measure theory of Lutz [18], [15] (also explained in §3 below). In the notation of this theory, our Main Theorem says that, for every real $\alpha < 1$,

$$\mu(\mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^c) \mid \mathrm{E}) = 0. \tag{1.1}$$

This means that $\mathrm{P}_{n^\alpha-\mathrm{tt}}(\mathrm{DENSE}^c) \cap \mathrm{E}$ is a *negligibly small* subset of E [18], [15].

In particular, our Main Theorem implies that

$$\mathrm{E} \not\subseteq \mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^c), \tag{1.2}$$

i.e., every $\leq_{n^\alpha-tt}^{\mathrm{P}}$-hard language for E is dense. This strengthens Theorem 2 by extending the truth-table reducibility from $O(\log n)$ queries to n^α queries ($\alpha < 1$). It is also worth noting that the combinatorial technique used to prove (1.1) and (1.2)—the *sequentially most frequent query selection*—is simpler than Watanabe's direct proof of Theorem 2. This is not surprising, once one considers that our proof of (1.2) via (1.1) is a resource-bounded instance of the *probabilistic method* [5], [25], [26], [6], [27], [1], which exploits the fact that it is often easier to prove the *abundance* of objects of a given type than to construct a *specific* object of that type.

Our proof of (1.1) also shows that, for every real $\alpha < 1$,

$$\mu(\mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^c) \mid \mathrm{E}_2) = 0. \tag{1.3}$$

Much of our interest in the Main Theorem concerns the class NP and Theorems 3 and 4. As already noted, for all reducibilities $\leq_r^{\mathrm{P}}$ discussed in this paper, all known $\leq_r^{\mathrm{P}}$-hard languages for NP are dense. One is thus led to ask whether there is a reasonable hypothesis θ such that we can prove results of the form

$$\theta \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_r(\mathrm{DENSE}^c), \tag{1.4}$$

for various choices of the reducibility $\leq_r^{\mathrm{P}}$. (Such a result is much stronger than the corresponding result

$$\theta \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_r(\mathrm{SPARSE}),$$

because there is an enormous gap between polynomial and 2^{n^ϵ} growth rates.)

Ogiwara and Watanabe's proof of Theorem 4 does not appear to allow significant relaxation of either the query bound or the sparseness criterion. In fact, it appears to be beyond current understanding to prove results of the form (1.4) if θ is "P $\neq$ NP." Karp and Lipton [12] have proven that

$$\Sigma_2^p \neq \Pi_2^p \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_T(\mathrm{SPARSE}).$$

That is, the stronger hypothesis $\Sigma_2^p \neq \Pi_2^p$ gives a stronger conclusion than those of Theorems 3 and 4. However, Karp and Lipton's proof does not appear to allow relaxation of the sparseness criterion, and results of the form (1.4) do not appear to be achievable at this time if θ is taken to be "$\Sigma_2^p \neq \Pi_2^p$."

To make progress on matters of this type, Lutz has proposed investigation of the measure-theoretic hypotheses $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$. These expressions say that NP does not have measure 0 in E_2 ("NP is not a negligible subset of E_2") and that NP does not have measure 0 in E ("NP $\cap$ E is not a negligible subset of E"), respectively. We now explain the meaning of these hypotheses. Both are best understood in terms of their negations.

The condition $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$ means that there exist a *fixed* polynomial q, a *fixed* positive quantity c_0 of capital (money), and a *fixed* betting strategy (algorithm) σ with the following properties. Given any language A, the strategy σ *bets* on the membership or nonmembership of the successive strings $\lambda, 0, 1, 00, 01, 10, \ldots$ in A. Before the betting begins, σ has capital (money) c_0. When betting on a string $w \in \{0,1\}^*$, the strategy σ is given as input the string consisting of the successive bits $[\![v \in A]\!]$ for all strings v that precede w in the standard ordering of $\{0,1\}^*$. On this input, the strategy σ computes, in $\leq 2^{q(|w|)}$ steps, a fraction $r \in [-1, 1]$ of its current capital to bet that $w \in A$. If σ's capital prior to this bet is c, then σ's capital after the bet is $c(1+r)$ if $w \in A$ and $c(1-r)$ if $w \notin A$. (That is, the betting is fair.) Finally, the strategy σ is *successful*, in the sense that, for all $A \in \mathrm{NP}$, σ's capital diverges to $+\infty$ as the betting progresses through the successive strings $w \in \{0,1\}^*$.

Thus, *the condition* $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$ *asserts the existence of a fixed* $2^{q(n)}$*-time-bounded algorithm* for betting successfully on membership of strings in all languages in NP. If NP $\subseteq$

DTIME($2^{r(n)}$) for some fixed polynomial r, it is easy to devise such a strategy, so $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$. (In fact, $\mu(\mathrm{DTIME}(2^{r(n)}) \mid \mathrm{E}_2) = 0$ [18].) Conversely, if $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$, then NP is "nearly contained in some fixed DTIME($2^{q(n)}$)," in the sense that there is a fixed $2^{q(n)}$-time-bounded algorithm σ for successfully betting on all languages in NP.

There does not appear to be any a priori reason for believing that such a strategy σ exists, i.e., there does not appear to be any a priori reason for believing that $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$. Similarly, there does not appear to be any a priori reason for believing that $\mu(\mathrm{NP} \mid \mathrm{E}) = 0$. The hypotheses $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$ are thus *reasonable relative to our current knowledge.* (The hypothesis that the polynomial-time hierarchy separates into infinitely many levels enjoys a similar status. It *may* be false, but if it *is* false, then a very remarkable algorithm exists.) Lutz has conjectured that the conditions $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$ may be true.

It should be noted here that the conditions $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$ are merely the negations of $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) = 0$, respectively. For example, a resource-bounded extension of the classical Kolmogorov zero-one law [15] implies that exactly one of the following three conditions must hold:

(i) $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$.

(ii) $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 1$, i.e., $\mu(\mathrm{NP}^c \mid \mathrm{E}_2) = 0$.

(iii) NP is not measurable in E_2, i.e., the expression $\mu(\mathrm{NP} \mid \mathrm{E}_2)$ has no value.

The condition $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ thus means that (ii) or (iii) is true. It does not mean that $\mu(\mathrm{NP} \mid E_2)$ has some positive value. The analogous remark applies to the condition $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$.

At this time, we are unable to prove or disprove the widely believed conjectures $\mathrm{P} \neq \mathrm{NP}$, $\mathrm{NP} \neq \mathrm{E}_2$, and $\mathrm{E} \not\subseteq \mathrm{NP}$. This, together with the known implications [18]

$$\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0 \Longrightarrow \mathrm{P} \neq \mathrm{NP},$$

$$\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0 \Longrightarrow \mathrm{P} \neq \mathrm{NP},$$

$$\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0 \Longrightarrow \mathrm{NP} \neq \mathrm{E}_2,$$

$$\mu(\mathrm{NP} \mid \mathrm{E}) = 0 \Longrightarrow \mathrm{E} \not\subseteq \mathrm{NP},$$

means that we are currently unable to prove or disprove the statements $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$.

Thus, at present, we are interested in the conditions $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$, not as conjectures, but rather as *scientific hypotheses*, which may have more *explanatory power* than traditional complexity-theoretic hypotheses such as $\mathrm{P} \neq \mathrm{NP}$ or the separation of the polynomial-time hierarchy. Until such time as mathematical proof or refutation is available, the reasonableness (or unreasonableness) of such hypotheses can be illuminated only by investigation of their *consequences*. Such investigation may indicate, for example, that the consequences of $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ form, en masse, a credible state of affairs, thereby increasing the reasonableness of this hypothesis. On the other hand, such investigation may uncover implausible consequences of $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$, or even a proof that $\mu(\mathrm{NP} \mid \mathrm{E}_2) = 0$. Either outcome would contribute to our understanding of NP.

Our Main Theorem implies that, for all $\alpha < 1$,

$$\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0 \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^c) \tag{1.5}$$

and

$$\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0 \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^c). \tag{1.6}$$

(This is Theorem 16.) That is, each of the hypotheses $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$ implies that every $\leq^{\mathrm{P}}_{n^\alpha-tt}$-hard language for NP is dense. This conclusion, which is credible and consistent with all observations to date, is not known to follow from $\mathrm{P} \neq \mathrm{NP}$ or other traditional complexity-theoretic hypotheses.

Recent investigation has also shown that the hypotheses $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$ and $\mu(\mathrm{NP} \mid \mathrm{E}) \neq 0$ imply that NP contains P-bi-immune languages [21] and that every $\leq^{\mathrm{P}}_{m}$-hard language for NP has an exponentially dense, exponentially hard complexity core [10]. Taken together, such results appear to indicate that these are reasonable hypotheses that may have considerable explanatory power.

The proof of our Main Theorem is based on a very general result, the Weak Stochasticity Theorem proven in §3. In very brief terms, this result says that almost every language in E, and almost every language in E_2, is "weakly stochastic," in the sense that it is statistically unpredictable by feasible deterministic algorithms, even with linear nonuniform advice. (See §3 for precise definitions.) This result enables us to prove our Main Theorem, that

$$\mu(\mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^{\mathrm{c}}) \mid \mathrm{E}) = \mu(\mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^{\mathrm{c}}) \mid \mathrm{E}_2) = 0$$

for all $\alpha < 1$, by a simple combinatorial technique, without reference to measure-theoretic notions. Specifically, in §4, this combinatorial technique—the *sequentially most frequent query selection*—is introduced and used to prove that *no* language in $\mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^{\mathrm{c}})$ is weakly stochastic. The Main Theorem follows immediately from this by the Weak Stochasticity Theorem.

This use of weak stochasticity in E and E_2 is analogous to earlier uses of space-bounded Kolmogorov complexity in the complexity class ESPACE $= \mathrm{DSPACE}(2^{\mathrm{linear}})$. It is known [18], [11] that almost every language in ESPACE has very high space-bounded Kolmogorov complexity. Using this fact, a variety of sets X have been shown to have measure 0 in ESPACE, simply by proving that every element of X has low space-bounded Kolmogorov complexity [18], [17], [19], [11]. Informally, we say that high space-bounded Kolmogorov complexity is a "general-purpose randomness property of languages in ESPACE." This expression, which is heuristic, means the following two things.

(a) Almost every language in ESPACE has the property (high space-bounded Kolmogorov complexity).

(b) It is often the case that, when one wants to prove a result of the form $\mu(X \mid \mathrm{ESPACE}) = 0$, it is convenient to prove that no language in X has the property and then appeal to (a).

It is natural to hope that high time-bounded Kolmogorov complexity would be, in the analogous sense, a general-purpose randomness property of language in E and E_2. Unfortunately, however, the strongest known lower bounds on time-bounded Kolmogorov complexity in these classes [18] are far too weak to provide a useful time-bounded analogue of condition (a) above. Moreover, improving these bounds appears to require a major breakthrough in complexity theory.

Our results suggest that, even without such a breakthrough, weak stochasticity may be a "general-purpose randomness property of languages in E and E_2." This would entail the following two heuristic conditions.

(a′) Almost every language in E (respectively, in E_2) is weakly stochastic.

(b′) It is often the case that, when one wants to prove a result of the form $\mu(X \mid \mathrm{E}) = 0$ (respectively, $\mu(X \mid \mathrm{E}_2) = 0$), it is convenient to prove that no language in X is weakly stochastic and then appeal to (a′).

The Weak Stochasticity Theorem gives us condition (a′) immediately. The proof of our Main Theorem gives us the *instance* $X = \mathrm{P}_{n^\alpha-tt}(\mathrm{DENSE}^{\mathrm{c}})$ of condition (b′). It appears likely that more such instances will arise, i.e., that weak stochasticity is a general-purpose randomness property of languages in E and E_2 that will be useful in future investigations.

2. Preliminaries. In this paper, $[\![\psi]\!]$ denotes the *Boolean value* of the condition ψ, i.e.,

$$[\![\psi]\!] = \begin{cases} 1 & \text{if } \psi, \\ 0 & \text{if not } \psi. \end{cases}$$

All *languages* here are sets of binary strings, i.e., sets $A \subseteq \{0,1\}^*$. We identify each language A with its *characteristic sequence* $\chi_A \in \{0,1\}^\infty$ defined by

$$\chi_A = [\![s_0 \in A]\!][\![s_1 \in A]\!][\![s_2 \in A]\!]\ldots,$$

where $s_0 = \lambda$, $s_1 = 0$, $s_2 = 1$, $s_3 = 00, \ldots$ is the standard enumeration of $\{0,1\}^*$. Relying on this identification, the set $\{0,1\}^\infty$, consisting of all infinite binary sequences, will be regarded as the set of all languages.

For $w \in \{0,1\}^*$ and $x \in \{0,1\}^* \cup \{0,1\}^\infty$, we say that w is a *prefix* of x and write $w \sqsubseteq x$, if $x = wy$ for some $y \in \{0,1\}^* \cup \{0,1\}^\infty$. The *cylinder generated by* a string $w \in \{0,1\}^*$ is

$$\mathbf{C}_w = \{x \in \{0,1\}^\infty \mid w \sqsubseteq x\}.$$

Note that $\mathbf{C}_w$ is a set of languages. Note also that $\mathbf{C}_\lambda = \{0,1\}^\infty$, where λ denotes the empty string.

As noted in §1, we work with the exponential time complexity classes E = DTIME(2^{linear}) and E_2 = DTIME($2^{\text{polynomial}}$). It is well known that $P \subsetneqq E \subsetneqq E_2$, that $P \subseteq NP \subseteq E_2$, and that $NP \neq E$.

We let $\mathbf{D} = \{m2^{-n} \mid m \in \mathbf{Z}, n \in \mathbf{N}\}$ be the set of *dyadic rationals*. (These are rational numbers whose standard binary representations have finite length.) We also fix a one-to-one pairing function $\langle\, , \rangle$ from $\{0,1\}^* \times \{0,1\}^*$ onto $\{0,1\}^*$ such that the pairing function and its associated projections, $\langle x, y\rangle \mapsto x$ and $\langle x, y\rangle \mapsto y$, are computable in polynomial time.

Several functions in this paper are of the form $d : \mathbf{N}^k \times \{0,1\}^* \to Y$, where Y is $\mathbf{D}$ or $[0, \infty)$, the set of nonnegative real numbers. Formally, in order to have uniform criteria for their computational complexities, we regard all such functions as having domain $\{0,1\}^*$ and having codomain $\{0,1\}^*$ if $Y = \mathbf{D}$. For example, a function $d : \mathbf{N}^2 \times \{0,1\}^* \to \mathbf{D}$ is formally interpreted as a function $\tilde{d} : \{0,1\}^* \to \{0,1\}^*$. Under this interpretation, $d(i, j, w) = r$ means that $\tilde{d}(\langle 0^i, \langle 0^j, w\rangle\rangle) = u$, where u is a suitable binary encoding of the dyadic rational r.

For a function $d : \mathbf{N} \times X \to Y$ and $k \in \mathbf{N}$, we define the function $d_k : X \to Y$ by $d_k(x) = d(k, x) = d(\langle 0^k, x\rangle)$. We then regard d as a "uniform enumeration" of the functions $d_0, d_1, d_2, \ldots$. For a function $d : \mathbf{N}^n \times X \to Y$ ($n \geq 2$), we write $d_{k,l} = (d_k)_l$, and so on.

For a function $\delta : \{0,1\}^* \to \{0,1\}^*$ and $n \in \mathbf{N}$, we write δ^n for the n-fold composition of δ with itself.

Our proof of the Weak Stochasticity Theorem uses the following form of the Chernoff bound.

LEMMA 5 [4], [8]. *If $X_1, \ldots, X_N$ are independent 0-1-valued random variables with the uniform distribution, $S = X_1 + \cdots + X_N$, and $\epsilon > 0$, then*

$$\Pr\left[\left|S - \frac{N}{2}\right| \geq \frac{\epsilon N}{2}\right] \leq 2e^{-\epsilon^2 N/6}.$$

In particular, taking $\epsilon = 2/(j+1)$, where $j \in \mathbf{N}$,

$$\Pr\left[\left|S - \frac{N}{2}\right| \geq \frac{N}{j+1}\right] \leq 2e^{-N/2(j+1)^2}.$$

Proof. See [8]. □

3. Measure and weak stochasticity. In this section, after reviewing some fundamentals of measure in exponential-time complexity classes, we prove the Weak Stochasticity Theorem. This theorem will be useful in the proof of our main result in §4. We also expect it to be useful in future investigations of the measure structure of E and E_2.

Resource-bounded measure [18], [15] is a very general theory whose special cases include classical Lebesgue measure, the measure structure of the class REC of all recursive languages, and measure in various complexity classes. In this paper we are interested only in measure in E and E_2, so our discussion of measure is specific to these classes. The interested reader may consult §3 of [18] for more discussion and examples.

Throughout this section, we identify every language $A \subseteq \{0, 1\}^*$ with its characteristic sequence $\chi_A \in \{0, 1\}^\infty$, defined as in §2.

A *constructor* is a function $\delta : \{0, 1\}^* \to \{0, 1\}^*$ such that $x \underset{\neq}{\sqsubseteq} \delta(x)$ for all $x \in \{0, 1\}^*$. The *result* of a constructor δ (i.e., the *language constructed by* δ) is the unique language $R(\delta)$ such that $\delta^n(\lambda) \sqsubseteq R(\delta)$ for all $n \in \mathbf{N}$. (Recall that this means that each string $\delta^n(\lambda)$ is a prefix of the characteristic sequence of $R(\delta)$.) Intuitively, δ constructs $R(\delta)$ by starting with λ and then iteratively generating successively longer prefixes of $R(\delta)$. Given a set Δ of functions from $\{0, 1\}^*$ into $\{0, 1\}^*$, we write $R(\Delta)$ for the set of all languages $R(\delta)$ such that $\delta \in \Delta$ and δ is a constructor.

We first note that the exponential-time complexity classes E and E_2 can be characterized in terms of constructors.

Notation. The classes $p_1 = p$ and p_2, both consisting of functions $f : \{0, 1\}^* \to \{0, 1\}^*$, are defined as follows.

$$p_1 = p = \{f \mid f \text{ is computable in polynomial time}\},$$

$$p_2 = \{f \mid f \text{ is computable in } n^{(\log n)^{O(1)}} \text{ time}\}.$$

LEMMA 6 [16]. (1) $R(p) = E$. (2) $R(p_2) = E_2$.

Using Lemma 6, the measure structures of E and E_2 are now developed in terms of the classes p_i, for $i = 1, 2$.

DEFINITION. *A* density function *is a function $d : \{0, 1\}^* \to [0, \infty)$ satisfying*

$$d(w) \geq \frac{d(w0) + d(w1)}{2} \tag{3.1}$$

for all $w \in \{0, 1\}^*$. The global value *of a density function d is $d(\lambda)$. The* set covered by *a density function d is*

$$S[d] = \bigcup_{\substack{w \in \{0,1\}^* \\ d(w) \geq 1}} \mathbf{C}_w. \tag{3.2}$$

(*Recall that $\mathbf{C}_w = \{x \in \{0, 1\}^\infty \mid w \sqsubseteq x\}$ is the cylinder generated by w.*) *A density function d* covers *a set $X \subseteq \{0, 1\}^\infty$ if $X \subseteq S[d]$.*

Consider the random experiment in which a sequence $x \in \{0, 1\}^\infty$ is chosen by using an independent toss of a fair coin to decide each bit of x. Taken together, parts (3.1) and (3.2) of the above definition imply that $\Pr[x \in S[d]] \leq d(\lambda)$ in this experiment. Intuitively, given a set $X \subseteq S[d]$, a density function d simply "asserts" that $d(\lambda)$ is an upper bound for $\Pr[x \in X]$ and then "verifies" this assertion by providing additional details in the form of values $d(w)$ satisfying conditions (3.1) and (3.2).

More generally, we will be interested in "uniform systems" of density functions that are computable within some resource bound.

DEFINITION. *An n-dimensional* density system *(n-DS) is a function*

$$d : \mathbf{N}^n \times \{0, 1\}^* \to [0, \infty)$$

such that $d_{\vec{k}}$ is a density function for every $\vec{k} \in \mathbf{N}^n$. It is sometimes convenient to regard a density function as a 0-DS.

DEFINITION. *A* computation *of an n-DS d is a function $\widehat{d} : \mathbf{N}^{n+1} \times \{0, 1\}^* \to \mathbf{D}$ such that*

$$\left|\widehat{d}_{\vec{k},r}(w) - d_{\vec{k}}(w)\right| \leq 2^{-r}$$

for all $\vec{k} \in \mathbf{N}^n$, $r \in \mathbf{N}$, and $w \in \{0, 1\}^*$. For $i = 1, 2$, a p_i-computation *of an n-DS d is a computation $\widehat{d}$ of d such that $\widehat{d} \in \mathrm{p}_i$. An n-DS d is* p_i-computable *if there exists a p_i-computation $\widehat{d}$ of d.*

If d is an n-DS such that $d : \mathbf{N}^n \times \{0, 1\}^* \to \mathbf{D}$ and $d \in \mathrm{p}_i$, then d is trivially p_i-computable. This fortunate circumstance, in which there is no need to compute approximations, occurs frequently in practice. (Such applications typically do involve approximations, but these are "hidden" by invoking fundamental theorems whose proofs involve approximations.)

We now come to the key idea of resource-bounded measure theory.

DEFINITION. *A* null cover *of a set $X \subseteq \{0, 1\}^\infty$ is a 1-DS d such that, for all $k \in \mathbf{N}$, d_k covers X with global value $d_k(\lambda) \leq 2^{-k}$. For $i = 1, 2$, a* p_i-null cover *of X is a null cover of X that is p_i-computable.*

In other words, a null cover of X is a uniform system of density functions that cover X with rapidly vanishing global value. It is easy to show [15] that a set $X \subseteq \{0, 1\}^\infty$ has classical Lebesgue measure 0 (i.e., probability 0 in the above coin-tossing experiment) if and only if there exists a null cover of X.

DEFINITION. *A set X has* p_i-measure 0, *and we write $\mu_{\mathrm{p}_i}(X) = 0$, if there exists a p_i-null cover of X. A set X has* p_i-measure 1, *and we write $\mu_{\mathrm{p}_i}(X) = 1$, if $\mu_{\mathrm{p}_i}(X^c) = 0$.*

Thus a set X has p_i-measure 0 if p_i provides sufficient computational resources to compute uniformly good approximations to a system of density functions that cover X with rapidly vanishing global value.

We now turn to the internal measure structures of $\mathrm{E} = R(\mathrm{p})$ and $\mathrm{E}_2 = R(\mathrm{p}_2)$.

DEFINITION. *A set X has* measure 0 in $R(\mathrm{p}_i)$, *and we write $\mu(X \mid R(\mathrm{p}_i)) = 0$, if $\mu_{\mathrm{p}_i}(X \cap R(\mathrm{p}_i)) = 0$. A set X has* measure 1 in $R(\mathrm{p}_i)$, *and we write $\mu(X \mid R(\mathrm{p}_i)) = 1$, if $\mu(X^c \mid R(\mathrm{p}_i)) = 0$. If $\mu(X \mid R(\mathrm{p}_i)) = 1$, we say that* almost every *language in $R(\mathrm{p}_i)$ is in X.*

The following lemma is obvious but useful.

LEMMA 7. *For every set $X \subseteq \{0, 1\}^\infty$,*

$$\begin{tikzcd}
\mu_p(X) = 0 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \mu_{p_2}(X) = 0 \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \Pr[x \in X] = 0 \\
\mu(X|\mathrm{E}) = 0 & \mu(X|\mathrm{E}_2) = 0,
\end{tikzcd}$$

where the probability $\Pr[x \in X]$ is computed according to the random experiment in which a sequence $x \in \{0, 1\}^\infty$ is chosen probabilistically, using an independent toss of a fair coin to decide each bit of x.

Thus a proof that a set X has p-measure 0 gives information about the size of X in E, in E_2, and in $\{0, 1\}^\infty$.

It was noted in Lemma 7 that $\mu_{\mathrm{p}}(X) = 0$ implies $\mu_{\mathrm{p}_2}(X) = 0$. In fact, more is true.

LEMMA 8 [14]. *Let Z be the union of all sets X such that $\mu_p(X) = 0$. Then $\mu_{p_2}(Z) = \mu(Z \mid \mathrm{E}_2) = 0$.*

Lemma 8 is also called the Abundance Theorem, because it implies that almost every language $A \in \mathrm{E}_2$ is p-*random*, i.e., has the property that the singleton set $\{A\}$ does *not* have p-measure 0. The proof of Lemma 8 makes essential use of the fact that p_2 contains a universal function for p. It is *not* the case that $\mu_{\mathrm{p}}(Z) = 0$.

It is shown in [18] that these definitions endow E and E_2 with internal measure structure. Specifically, for $i = 1, 2$, if $\mathcal{I}$ is either the collection $\mathcal{I}_{\mathrm{p}_i}$ of all p_i-measure 0 sets or the collection $\mathcal{I}_{R(\mathrm{p}_i)}$ of all sets of measure 0 in $R(\mathrm{p}_i)$, then $\mathcal{I}$ is a "p_i-ideal," i.e., is closed under subsets, finite unions, and "p_i-unions" (countable unions that can be generated with the resources of p_i). More importantly, the Measure Conservation Theorem of [18] says that the ideal $\mathcal{I}_{R(\mathrm{p}_i)}$ is a *proper* ideal, i.e., that E does *not* have measure 0 in E and E_2 does *not* have measure 0 in E_2. Taken together, these facts justify the intuition that, if $\mu(X|\mathrm{E}) = 0$, then $X \cap \mathrm{E}$ is a *negligibly small* subset of E (and similarly for E_2).

Our proof of the Weak Stochasticity Theorem does not directly use the above definitions. Instead we use a sufficient condition, proved in [18], for a set to have measure 0. To state this condition we need a polynomial notion of convergence for infinite series. All our series here consist of nonnegative terms. A *modulus* for a series $\sum_{n=0}^{\infty} a_n$ is a function $m : \mathbf{N} \to \mathbf{N}$ such that

$$\sum_{n=m(j)}^{\infty} a_n \leq 2^{-j}$$

for all $j \in \mathbf{N}$. A series is p-*convergent* if it has a modulus that is a polynomial. A sequence

$$\sum_{k=0}^{\infty} a_{j,k} \qquad (j = 0, 1, 2, \ldots)$$

of series is *uniformly* p-*convergent* if there exists a polynomial $m : \mathbf{N}^2 \to \mathbf{N}$ such that, for each $j \in \mathbf{N}$, m_j is a modulus for the series $\sum_{k=0}^{\infty} a_{j,k}$. We will use the following sufficient condition for uniform p-convergence. (This well-known lemma is easily verified by routine calculus.)

LEMMA 9. *Let $a_{j,k} \in [0, \infty)$ for all $j, k \in \mathbf{N}$. If there exist a real $\epsilon > 0$ and a polynomial $g : \mathbf{N} \to \mathbf{N}$ such that $a_{j,k} \leq e^{-k^{\epsilon}}$ for all $j, k \in \mathbf{N}$ with $k \geq g(j)$, then the series*

$$\sum_{k=0}^{\infty} a_{j,k} \qquad (j = 0, 1, 2, \ldots)$$

are uniformly p-convergent.

The proof of the Weak Stochasticity Theorem is greatly simplified by using the following special case (for p) of a uniform, resource-bounded generalization of the classical first Borel-Cantelli lemma.

LEMMA 10 [18]. *If d is a p-computable 2-DS such that the series*

$$\sum_{k=0}^{\infty} d_{j,k}(\lambda) \qquad (j = 0, 1, 2, \ldots)$$

are uniformly p-convergent, then

$$\mu_p\left(\bigcup_{j=0}^{\infty} \bigcap_{t=0}^{\infty} \bigcup_{k=t}^{\infty} S[d_{j,k}]\right) = 0.$$

If we write $S_j = \bigcap_{t=0}^{\infty} \bigcup_{k=t}^{\infty} S[d_{j,k}]$ and $S = \bigcup_{j=0}^{\infty} S_j$, then Lemma 10 gives a sufficient condition for concluding that S has p-measure 0. Note that each S_j consists of those languages A that are in infinitely many of the sets $S[d_{j,k}]$.

We now formulate our notion of weak stochasticity. For this we need a few definitions. Our notion of advice classes is standard [12]. An *advice function* is a function $h : \mathbf{N} \to \{0, 1\}^*$. Given a function $q : \mathbf{N} \to \mathbf{N}$, we write ADV$(q)$ for the set of all advice functions h such that $|h(n)| \leq q(n)$ for all $n \in \mathbf{N}$. Given a language $A \subseteq \{0, 1\}^*$ and an advice function h, we define the language A/h ("A with advice h") by

$$A/h = \{x \in \{0, 1\}^* \mid \langle x, h(|x|)\rangle \in A\}.$$

Given functions $t, q : \mathbf{N} \to \mathbf{N}$, we define the *advice class*

$$\mathrm{DTIME}(t)/\mathrm{ADV}(q) = \{A/h \mid A \in \mathrm{DTIME}(t), h \in \mathrm{ADV}(q)\}.$$

DEFINITION. *Let $t, q, \nu : \mathbf{N} \to \mathbf{N}$ and let $A \subseteq \{0, 1\}^*$. Then A is* weakly (t, q, ν)-stochastic *if, for all $B \in \mathrm{DTIME}(t)/\mathrm{ADV}(q)$ and all $C \in \mathrm{DTIME}(t)$ such that $|C_{=n}| \geq \nu(n)$ for all sufficiently large n,*

$$\lim_{n\to\infty} \frac{|(A \triangle B) \cap C_{=n}|}{|C_{=n}|} = \frac{1}{2}.$$

Intuitively, B and C together form a "prediction scheme" in which B tries to guess the behavior of A on the set C. A is weakly (t, q, ν)-stochastic if no such scheme is better in the limit than guessing by random tosses of a fair coin.

Our use of the term "stochastic" follows Kolmogorov's terminology [13], [29] for properties defined in terms of limiting frequencies of failure of prediction schemes. The adverb "weakly" distinguishes our notion from a stronger stochasticity property considered in [14], but weak stochasticity is a powerful and convenient tool.

The following lemma is central to this section because it captures the main technical content of the Weak Stochasticity Theorem.

LEMMA 11. *Fix $c \in \mathbf{N}$ and $0 < \gamma \in \mathbf{R}$, and let*

$$WS_{c,\gamma} = \{A \subseteq \{0, 1\}^* | A \textit{ is weakly } (2^{cn}, cn, 2^{\gamma n})\textit{-stochastic}\}.$$

Then $\mu_{\mathrm{p}}(WS_{c,\gamma}) = 1$.

Proof. Assume the hypothesis. Let $U \in \mathrm{DTIME}(2^{(c+1)n})$ be a language that is universal for $\mathrm{DTIME}(2^{cn}) \times \mathrm{DTIME}(2^{cn})$ in the following sense. For each $i \in \mathbf{N}$, let

$$C_i = \{x \in \{0, 1\}^* | \langle 0^i, 0x\rangle \in U\},$$

$$D_i = \{x \in \{0, 1\}^* | \langle 0^i, 1x\rangle \in U\}.$$

Then $\mathrm{DTIME}(2^{cn}) \times \mathrm{DTIME}(2^{cn}) = \{(C_i, D_i) | i \in \mathbf{N}\}$.

Our objective is to use Lemma 10 to prove that $WS^{c}_{c,\gamma}$, the complement of $WS_{c,\gamma}$, has p-measure 0. In order to do this, for all $i, j, k \in \mathbf{N}$, we define the set $Y_{i,j,k}$ of languages as follows. If k is not a power of 2, then $Y_{i,j,k} = \emptyset$. Otherwise, if $k = 2^n$, where $n \in \mathbf{N}$, then

$$Y_{i,j,k} = \bigcup_{z \in \{0,1\}^{\leq cn}} Y_{i,j,k,z},$$

where each

$$Y_{i,j,k,z} = \left\{ A \subseteq \{0,1\}^* \;\middle|\; |(C_i)_{=n}| \geq 2^{\gamma n} \right.$$
$$\left. \text{and} \left| \frac{|(A \bigtriangleup (D_i/z)) \cap (C_i)_{=n}|}{|(C_i)_{=n}|} - \frac{1}{2} \right| \geq \frac{1}{j+1} \right\}.$$

(The notation D_i/z here denotes D_i/h, where $h : \mathbf{N} \to \{0,1\}^*$ is the constant function $h(n) = z$.) The point of this definition is that, if a language $A \subseteq \{0,1\}^*$ is *not* an element of $WS_{c,\gamma}$, then the definition of weak stochasticity says that there exist $i, j \in \mathbf{N}$ such that $A \in Y_{i,j,k}$ for infinitely many k. That is,

$$WS^c_{c,\gamma} \subseteq \bigcup_{i=0}^{\infty} \bigcup_{j=0}^{\infty} \bigcap_{m=0}^{\infty} \bigcup_{k=m}^{\infty} Y_{i,j,k}.$$

It follows by Lemma 10 that it suffices to exhibit a p-computable 3-*DS* d with the following two properties.

(I) The series $\sum_{k=0}^{\infty} d_{i,j,k}(\lambda)$, for $i, j \in \mathbf{N}$, are uniformly p-convergent.

(II) For all $i, j, k \in \mathbf{N}$, $Y_{i,j,k} \subseteq S[d_{i,j,k}]$.

Define the function $d : \mathbf{N}^3 \times \{0,1\}^* \to [0,\infty)$ as follows. If k is not a power of 2, then $d_{i,j,k}(w) = 0$. Otherwise, if $k = 2^n$, where $n \in \mathbf{N}$, then

$$d_{i,j,k}(w) = \sum_{z \in \{0,1\}^{\leq cn}} \Pr(Y_{i,j,k,z} | \mathbf{C}_w),$$

where the conditional probabilities $\Pr(Y_{i,j,k,z}|\mathbf{C}_w) = \Pr[A \in Y_{i,j,k,z} | A \in \mathbf{C}_w]$ are computed according to the random experiment in which the language $A \subseteq \{0,1\}^*$ is chosen probabilistically, using an independent toss of a fair coin to decide membership of each string in A.

It follows immediately from the definition of conditional probability that d is a 3-*DS*. Since $U \in \mathrm{DTIME}(2^{(c+1)n})$ and c is fixed, we can use binomial coefficients to compute (exactly) $d_{i,j,k}(w)$ in time polynomial in $i + j + k + |w|$. Thus d is p-computable.

To see that d has property (I), note first that the Chernoff bound, Lemma 5, tells us that, for all $i, j, k \in \mathbf{N}$ and $z \in \{0,1\}^{\leq cn}$ (writing $k = 2^n$ and $N = k^\gamma = 2^{\gamma n}$),

$$\Pr(Y_{i,j,k,z}) \leq 2e^{-N/2(j+1)^2},$$

whence

$$\begin{aligned} d_{i,j,k}(\lambda) &= \sum_{z \in \{0,1\}^{\leq cn}} \Pr(Y_{i,j,k,z}) \\ &\leq 2^{cn+1} \cdot 2e^{-N/2(j+1)^2} \\ &< e^{cn+2-N/2(j+1)^2}. \end{aligned}$$

Let $a = \lceil \frac{1}{\gamma} \rceil$, let $\delta = \frac{\gamma}{4}$, and fix $k_0 \in \mathbf{N}$ such that

$$k^{2\delta} \geq k^{\delta} + c \log k + 2$$

for all $k \geq k_0$. Define $g : \mathbf{N} \to \mathbf{N}$ by

$$g(j) = 4^a (j+1)^{4a} + k_0.$$

Then g is a polynomial and, for all $i, j, n \in \mathbf{N}$ (writing $k = 2^n$ and $N = k^\gamma = k^{4\delta}$),

$$k \geq g(j) \implies \left\{ \begin{array}{rcl} N & = & k^{2\delta}k^{2\delta} \\ & \geq & [4^\alpha(j+1)^{4\alpha}]^{2\delta}(k^\delta + c\log k + 2) \\ & \geq & 2(j+1)^2(k^\delta + cn + 2) \end{array} \right.$$
$$\implies d_{i,j,k}(\lambda) < e^{-k^\delta}.$$

Thus $d_{i,j,k}(\lambda) < e^{-k^\delta}$ for all $i, j, k \in \mathbf{N}$ such that $k \geq g(j)$. Since $\delta > 0$, it follows by Lemma 9 that (I) holds.

Finally, to see that (II) holds, fix $i, j, k \in \mathbf{N}$. If k is not a power of 2, then (II) is trivially affirmed, so assume that $k = 2^n$, where $n \in \mathbf{N}$. Let $A \in Y_{i,j,k}$. Fix $z \in \{0, 1\}^{\leq cn}$ such that $A \in Y_{i,j,k,z}$ and let w be the $(2^{n+1} - 1)$-bit characteristic string of $A_{\leq n}$. Then

$$d_{i,j,k}(w) \geq \Pr(Y_{i,j,k,z}|\mathbf{C}_w) = 1,$$

so $A \in \mathbf{C}_w \subseteq S[d_{i,j,k}]$. This completes the proof of Lemma 11. □

We now have the main result of this section.

THEOREM 12 (Weak Stochasticity Theorem). (1) *For all* $c \in \mathbf{N}$ *and* $\gamma > 0$, *almost every language* $A \in \mathrm{E}$ *is weakly* $(2^{cn}, cn, 2^{\gamma n})$*-stochastic.* (2) *Almost every language* $A \in \mathrm{E}_2$ *is, for all* $c \in \mathbf{N}$ *and* $\gamma > 0$, *weakly* $(2^{cn}, cn, 2^{\gamma n})$*-stochastic.*

Proof. Part (1) follows immediately from Lemma 11 via Lemma 7. Part (2) follows from Lemma 11 via Lemmas 8 and 7. □

4. The density of hard languages. In this section we prove our main result, that for every real $\alpha < 1$, the set $\mathrm{P}_{n^\alpha-\mathrm{tt}}(\mathrm{DENSE}^c)$ has measure 0 in E and in E_2. We then derive some consequences of this result. Some terminology and notation will be useful.

Given a query-counting function $q : \mathbf{N} \to \mathbf{Z}^+$, a $q(n)$*-query function* is a function f with domain $\{0, 1\}^*$ such that, for all $x \in \{0, 1\}^*$,

$$f(x) = (f_1(x), \ldots, f_{q(|x|)}(x)) \in (\{0, 1\}^*)^{q(|x|)}.$$

Each $f_i(x)$ is called a *query* of f on input x. A $q(n)$*-truth-table function* is a function g with domain $\{0, 1\}^*$ such that, for each $x \in \{0, 1\}^*$, $g(x)$ is the encoding of a $q(|x|)$-input, 1-output Boolean circuit. We write $g(x)(w)$ for the output of this circuit on input $w \in \{0, 1\}^{q(|x|)}$. A $\leq^{\mathrm{P}}_{q(n)-tt}$*-reduction* is an ordered pair (f, g) such that f is a $q(n)$-query function, g is a $q(n)$-truth-table function, and f and g are computable in polynomial time.

Let $A, B \subseteq \{0, 1\}^*$. A $\leq^{\mathrm{P}}_{q(n)-tt}$*-reduction of* A *to* B is a $\leq^{\mathrm{P}}_{q(n)-tt}$-reduction (f, g) such that, for all $x \in \{0, 1\}^*$,

$$[\![x \in A]\!] = g(x)([\![f_1(x) \in B]\!] \cdots [\![f_{q(|x|)}(x) \in B]\!]).$$

(Recall that $[\![\psi]\!]$ denotes the Boolean value of the condition ψ.) In this case we say that $A \leq^{\mathrm{P}}_{q(n)-tt} B$ via (f, g). We say that A is $\leq^{\mathrm{P}}_{q(n)-tt}$*-reducible to* B and write $A \leq^{\mathrm{P}}_{q(n)-tt} B$, if there exists (f, g) such that $A \leq^{\mathrm{P}}_{q(n)-tt} B$ via (f, g).

The proof of our main result makes essential use of the following construction.

Given an n^α-query function f and $n \in \mathbf{N}$, the *sequentially most frequent query selection* (*smfq selection*) for f on inputs of length n is the sequence

$$(S_0, Q_0, y_0), (S_1, Q_1, y_1), \ldots, (S_{n^\alpha}, Q_{n^\alpha}, y_{n^\alpha})$$

defined as follows. Each $S_k \subseteq \{0, 1\}^n$. Each Q_k is an $|S_k| \times n^\alpha$ matrix of strings, with each string in Q_k colored either green or red. The rows of Q_k are indexed lexicographically by the elements of S_k. For $x \in S_k$, row x of Q_k is the sequence $f_1(x), \ldots, f_{n^\alpha}(x)$ of queries of f on input x. If Q_k contains at least one green string, then y_k is the green string occurring in the greatest number of rows of Q_k. (Ties are broken lexicographically.) If Q_k is entirely red, then $y_k = \top$ ("top," i.e., undefined). The sets S_k and the coloring are specified recursively. We set $S_0 = \{0, 1\}^n$ and color all strings in Q_0 green. Assume that S_k, Q_k, and y_k have been defined, where $0 \leq k < n^\alpha$. If $y_k = \top$, then $(S_{k+1}, Q_{k+1}, y_{k+1}) = (S_k, Q_k, y_k)$. If $y_k \neq \top$, then S_{k+1} is the set of all $x \in S_k$ such that y_k appears in row x of Q_k. The strings in Q_{k+1} are then colored exactly as they were in Q_k, except that all y_k's are now colored red. This completes the definition of the smfq selection.

For $0 \leq k \leq n^\alpha$, it is clear that every row of Q_k contains at least k red strings. In particular, the matrix Q_{n^α} is entirely red.

Our main results follow from the following lemma. Recall that $WS_{c,\gamma}$ is the set of all weakly $(2^{cn}, cn, 2^{\gamma n})$-stochastic languages.

LEMMA 13. *For every real* $\alpha < 1$, $\mathrm{P}_{n^\alpha\text{-tt}}(\mathrm{DENSE}^c) \cap WS_{3,1/2} = \emptyset$.

Proof. Let $\alpha < 1$, and assume that $A \leq^{\mathrm{P}}_{n^\alpha-tt} L$ via (f, g), where $L \notin$ DENSE. It suffices to show that $A \notin WS_{3,1/2}$. Fix a polynomial p such that $|f_i(x)| \leq p(|x|)$ for all $x \in \{0, 1\}^*$ and $1 \leq i \leq |x|^\alpha$. Let $\epsilon = (1 - \alpha)/4$, and fix $n_0 \in \mathbf{N}$ such that the following conditions hold for all $n \geq n_0$:

(i) $n \geq 2 \cdot n^{1-2\epsilon}$,

(ii) $n^{2\epsilon} - n^\epsilon \geq 2$.

Let

$$K = \{n \in \mathbf{N} \mid n \geq n_0 \text{ and } |L_{\leq p(n)}| < 2^{n^\epsilon}\}.$$

Note that K is infinite because L is not dense.

Define languages B, C, D and an advice function $h : \mathbf{N} \to \{0, 1\}^*$ as follows. For all $n < n_0$, we let $C_{=n} = \{0, 1\}^n$, $h(n) = \lambda$, and $D_n = \{\langle x, \lambda\rangle \mid x \in \{0, 1\}^n\}$. For all $n \geq n_0$, $C_{=n}$, $h(n)$, and D_n are defined from the smfq selection for f on inputs of length n as follows: Let $k = k(n)$ be the greatest integer such that $0 \leq k \leq n^\alpha$ and $|S_k| \geq 2^{n-kn^{2\epsilon}}$. (Note that k exists because $|S_0| = 2^n$.) We then define

$$C_{=n} = S_k,$$

$$h(n) = [\![y_0 \in L]\!] \cdots [\![y_{k-1} \in L]\!],$$

and we let D_n be the set of all coded pairs $\langle x, z\rangle$ such that $x \in S_k$, $z \in \{0, 1\}^k$, and $g(x)(b_1 \ldots b_{n^\alpha}) = 1$, where each

$$b_i = \begin{cases} z[j] & \text{if } f_i(x) = y_j, 0 \leq j < k, \\ 0 & \text{if } f_i(x) \notin \{y_0, \ldots, y_{k-1}\}. \end{cases}$$

Finally, we let $D = \bigcup_{n=0}^{\infty} D_n$ and $B = D/h$. Intuitively here, B tries to predict A on C. Specifically, for each $n \geq n_0$ and each $x \in C_{=n} = S_k$, the bit $[\![x \in B]\!]$ is a "guessed value" of the bit $[\![x \in A]\!]$. The actual value, given by the reduction (f, g) to L, is

$$[\![x \in A]\!] = g(x)([\![w_1 \in L]\!] \cdots [\![w_{n^\alpha} \in L]\!]),$$

where $w_1, \ldots, w_{n^\alpha}$ are the entries in row x of the matrix Q_k. The guessed value $[\![x \in B]\!] = g(x)(b_1 \ldots b_{n^\alpha})$ uses the advice function h to get the *correct* bit $b_i = [\![w_i \in L]\!]$ when the string w_i is red in Q_k, and *guesses* that $w_i \notin L$ when the string w_i is green in Q_k.

It is easy to see that $C, D \in \mathrm{DTIME}(2^{3n})$ and $B \in \mathrm{DTIME}(2^{3n})/\mathrm{ADV}(3n)$. (The bound $3n$ is generous here.) Also, by condition (i) in our choice of n_0,

$$|C_{=n}| \geq 2^{n-n^{\alpha}n^{2\epsilon}} \geq 2^{n/2}$$

for all $n \geq n_0$, whence $|C_{=n}| \geq 2^{\frac{n}{2}}$ for all $n \in \mathbf{N}$.

We now show that B does a good job of predicting A on $C_{=n}$, for all $n \in K$. Let $n \in K$. We have two cases.

(I) If $k = k(n) = n^{\alpha}$, then all strings in Q_k are red, so *all* the guesses made by B are correct, so

$$|(A \triangle B) \cap C_{=n}| = 0.$$

(II) If $k = k(n) < n^{\alpha}$, let r be the number of rows in Q_k, i.e., $r = |S_k| = |C_{=n}|$. By our choice of k, we have

$$|S_{k+1}| \leq 2^{n-(k+1)n^{2\epsilon}} \leq 2^{-n^{2\epsilon}}r.$$

That is, no green string appears in more than $2^{-n^{2\epsilon}}r$ of the rows of Q_k. Moreover, since $|L_{\leq p(n)}| \leq 2^{n^{\epsilon}}$, there are at most $2^{n^{\epsilon}}$ green strings w in Q_k such that $w \in L$. Thus there are at most $2^{n^{\epsilon}} \cdot 2^{-n^{2\epsilon}}r = 2^{n^{\epsilon}-n^{2\epsilon}}r$ rows of Q_k in which B makes an incorrect guess that a green string is not in L; the guesses made by B are correct in all other rows! By condition (ii) in our choice of n_0, then, B is incorrect in at most $\frac{1}{4}r$ rows of Q_k. That is,

$$|(A \triangle B) \cap C_{=n}| \leq \tfrac{1}{4}r.$$

In either case, (I) or (II), we have

$$|(A \triangle B) \cap C_{=n}| \leq \tfrac{1}{4}|C_{=n}|.$$

Since this holds for all $n \in K$, and since K is infinite,

$$\frac{|(A \triangle B) \cap C_{=n}|}{|C_{=n}|} \not\to \frac{1}{2}.$$

Thus B and C testify that A is not weakly $(2^{3n}, 3n, 2^{n/2})$-stochastic, i.e., that $A \notin WS_{3,1/2}$. □

Our main results are now easily derived. We start with the fact that most languages decidable in exponential time are not $\leq^{\mathrm{P}}_{n^{\alpha}-tt}$-reducible to nondense languages.

THEOREM 14 (Main Theorem). *For every real number* $\alpha < 1$,

$$\mu(\mathrm{P}_{n^{\alpha}-tt}(\mathrm{DENSE}^{\mathrm{c}}) \mid \mathrm{E}) = \mu(\mathrm{P}_{n^{\alpha}-tt}(\mathrm{DENSE}^{\mathrm{c}}) \mid \mathrm{E}_2) = 0.$$

Proof. This follows immediately from Theorem 12 and Lemma 13. □

The Main Theorem yields the following separation result.

THEOREM 15. *For every real* $\alpha < 1$,

$$\mathrm{E} \not\subseteq \mathrm{P}_{n^{\alpha}-tt}(\mathrm{DENSE}^{\mathrm{c}}).$$

That is, every $\leq^{\mathrm{P}}_{n^{\alpha}-tt}$*-hard language for* E *is dense.*

Proof. By the Measure Conservation Theorem [18], $\mu(\mathrm{E} \mid \mathrm{E}) \neq 0$, so this follows immediately from Theorem 14. □

Note that Theorem 15 strengthens Theorem 2 by extending the number of queries from $O(\log n)$ to n^{α}, where $\alpha < 1$ (e.g., $\alpha = 0.99$).

It is worthwhile to examine the roles played by various methods. Theorem 14, a measure-theoretic result concerning the *quantitative* structure of E and E_2, yields the *qualitative* separation result Theorem 15. From a technical standpoint, this proof of Theorem 15 has the following three components.

(i) The sequentially most frequent query selection (Lemma 13). This is used to prove that every language in $P_{n^{\alpha}-tt}(\mathrm{DENSE}^c)$ is predictable, i.e., fails to be weakly stochastic (with suitable parameters).

(ii) The Weak Stochasticity Theorem (Theorem 12). This shows that only a measure 0 subset of the languages in E are predictable.

(iii) The Measure Conservation Theorem [18]. This shows that E is not a measure 0 subset of itself.

Of these three components, (ii) and (iii) are general theorems concerning measure in E. Only component (i) is specific to the issue of the densities of $P_{n^{\alpha}-tt}$-hard languages. That is, *given the general principles* (ii) and (iii), the proof of Theorem 15 is just the sequentially most frequent query selection, i.e., the proof of Lemma 13. The latter proof is combinatorially much simpler than Watanabe's direct proof of Theorem 2. This is not surprising, once it is noted that our proof of Theorem 15 is an application of (a resource-bounded generalization of) the *probabilistic method* [5], [25], [26], [6], [27], [1], which exploits the fact that it is often easier to establish the *abundance* of objects of a given type than to construct a *specific* object of that type. Much of our proof of Theorem 15 is "hidden" in the power of this method (i.e., in the proofs of the Measure Conservation and Weak Stochasticity Theorems), freeing us to apply the sequentially most frequent query selection to the problem at hand.

An important feature of this general method is that it is *uniformly constructive* in the following sense. Taken together, the proofs of the Measure Conservation and Weak Stochasticity Theorems give a straightforward, "automatic" construction of a language $A \in \mathrm{E} \cap \mathrm{WS}_{3,1/2}$. By Lemma 13, it follows immediately that $A \in \mathrm{E} \backslash P_{n^{\alpha}-tt}(\mathrm{DENSE}^c)$. Thus one can apply this complexity-theoretic version of the probabilistic method with complete assurance that the resulting existence proof will automatically translate into a construction.

The primary objective of resource-bounded measure theory is to give a detailed account of the *quantitative structure* of E, E_2, and other complexity classes. The derivation of *qualitative* separation results, such as Theorems 15 and 2, is only a by-product of this quantitative objective. (By analogy, the value of classical Lebesgue measure and probability far surpasses their role as tools for existence proofs.) In the case of E, for example, the quantitative content of Theorem 14 is that the set $P_{n^{\alpha}-tt}(\mathrm{DENSE}^c) \cap \mathrm{E}$ is a *negligibly small* subset of E.

As noted in the introduction to this paper, we are interested in the consequences of the hypothesis that NP is *not* a negligibly small subset of exponential time. In this regard, our main theorem yields the following result.

THEOREM 16. *If* $\mu(\mathrm{NP}|\mathrm{E}) \neq 0$ *or* $\mu(\mathrm{NP}|\mathrm{E}_2) \neq 0$, *then for all* $\alpha < 1$, *every* $\leq^{\mathrm{P}}_{n^{\alpha}-tt}$*-hard language for* NP *is dense, i.e.,* $\mathrm{NP} \not\subseteq P_{n^{\alpha}-\mathrm{tt}}(\mathrm{DENSE}^c)$.

Proof. If NP has a $\leq^{\mathrm{P}}_{n^{\alpha}-tt}$-hard language H that is not dense then Theorem 14 tells us that $\mu(\mathrm{NP}|\mathrm{E}) = \mu(P_{n^{\alpha}-\mathrm{tt}}(H)|\mathrm{E}) = 0$ and $\mu(\mathrm{NP}|\mathrm{E}_2) = \mu(P_{n^{\alpha}-\mathrm{tt}}(H)|\mathrm{E}_2) = 0$. □

Note that the hypothesis and conclusion of Theorem 16 are both stronger than their counterparts in Ogiwara and Watanabe's result that

$$\mathrm{P} \neq \mathrm{NP} \Rightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{\mathrm{btt}}(\mathrm{SPARSE}).$$

Note also that our proof of Theorem 16 actually shows that

$$\mathrm{NP} \cap \mathrm{WS}_{3,1/2} \neq \emptyset \Longrightarrow \mathrm{NP} \not\subseteq P_{n^{\alpha}-tt}(\mathrm{DENSE}^c).$$

In fact, this implication and Theorem 16 both hold with NP replaced by PH, PP, PSPACE, or any other class.

5. Conclusion. The density criterion in Theorem 14 cannot be improved, since for every $\epsilon > 0$ there is a language $A \in \mathrm{E}$ that is $\leq^{\mathrm{P}}_{m}$-hard for E_2 and satisfies $|A_{\leq n}| < 2^{n^{\epsilon}}$ for all n. It is an open question whether the query bound n^{α} can be significantly relaxed. A construction of Wilson [32] shows that there is an oracle B such that $\mathrm{E}^B \subseteq \mathrm{P}^B_{O(n)-\mathrm{tt}}(\mathrm{SPARSE})$, so progress in this direction will require nonrelativizable techniques. (The proof of Theorem 14 relativizes in a straightforward manner.)

There are several open questions involving special reducibilities. We mention just one example. Very recently, Arvind, Köbler, and Mundhenk [2] proved that

$$\mathrm{P} \neq \mathrm{NP} \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{btt}(\mathrm{P}_{ctt}(\mathrm{SPARSE})),$$

where P_{ctt} refers to polynomial-time *conjunctive* reducibility. (This strengthens Theorem 4.) Does the class $\mathrm{P}_{btt}(\mathrm{P}_{ctt}(\mathrm{DENSE}^{\mathrm{c}}))$ have measure 0 in E?

As noted in the introduction, all known $\leq^{\mathrm{P}}_{\mathrm{T}}$-hard languages for NP are dense, i.e., our experience suggests that $\mathrm{NP} \not\subseteq \mathrm{P}_T(\mathrm{DENSE}^{\mathrm{c}})$. This suggests two open questions. (See Fig. 1.) Karp and Lipton [12] show that

$$\begin{tikzcd}
 & \Sigma^{\mathrm{p}}_2 \neq \Pi^{\mathrm{p}}_2 \arrow[dl] \arrow[dd, "\text{Karp and Lipton}"] & & \mu(\Sigma^{\mathrm{p}}_2 \setminus \Pi^{\mathrm{p}}_2 \mid \mathrm{E}_2) \neq 0 \arrow[lll] \arrow[dl, dashed, "\text{open}"] \arrow[dd, dashed, "\text{open}"] \\
\mathrm{P} \neq \mathrm{NP} \arrow[dd, "\text{Ogiwara and Watanabe}"] & & \mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0 \arrow[ll] \arrow[dd, "\text{this paper}"] & \\
 & \mathrm{NP} \not\subseteq \mathrm{P}_{\mathrm{T}}(\mathrm{SPARSE}) \arrow[dl] & & \mathrm{NP} \not\subseteq \mathrm{P}_{\mathrm{T}}(\mathrm{DENSE}^{\mathrm{c}}) \arrow[ll] \arrow[dl] \\
\mathrm{NP} \not\subseteq \mathrm{P}_{\mathrm{btt}}(\mathrm{SPARSE}) & & \mathrm{NP} \not\subseteq \mathrm{P}_{\mathrm{n}^{\alpha}\text{-}\mathrm{tt}}(\mathrm{DENSE}^{\mathrm{c}}) \arrow[ll] &
\end{tikzcd}$$

Fig. 1. *Two open problems.*

$$\Sigma^p_2 \neq \Pi^p_2 \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_T(\mathrm{SPARSE}).$$

Theorem 16 of the present paper shows that

$$\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0 \Longrightarrow \mathrm{NP} \not\subseteq \mathrm{P}_{n^{\alpha}-tt}(\mathrm{DENSE}^{\mathrm{c}})$$

for $\alpha < 1$. The first question, posed by Selman [24], is whether the strong hypothesis $\mu(\Sigma^p_2 \backslash \Pi^p_2 \mid \mathrm{E}_2) \neq 0$ can be used to combine these ideas to get a conclusion that $\mathrm{NP} \not\subseteq \mathrm{P}_T(\mathrm{DENSE}^{\mathrm{c}})$. The second, more fundamental, question is suggested by the first. A well-known downward separation principle [28] says that, if the polynomial time hierarchy separates at some level, then it separates at all lower levels. Thus, for example, $\Sigma^p_2 \neq \Pi^p_2$ implies that

P $\neq$ NP. Is there a "downward measure separation principle," stating that $\mu(\Sigma_{k+1}^p \backslash \Pi_{k+1}^p \mid \mathrm{E}_2) \neq 0 \Longrightarrow \mu(\Sigma_k^p \backslash \Pi_k^p \mid \mathrm{E}_2) \neq 0$? In particular, does $\mu(\Sigma_2^p \backslash \Pi_2^p \mid \mathrm{E}_2) \neq 0$ imply that $\mu(\mathrm{NP} \mid \mathrm{E}_2) \neq 0$?

The hypothesis that $\mu(\mathrm{NP}|\mathrm{E}_2) \neq 0$, i.e., that NP is not a negligibly small subset of E_2, was recently shown to have a number of credible consequences: If $\mu(\mathrm{NP}|\mathrm{E}_2) \neq 0$, then NP contains p-random languages [14]; NP contains E-bi-immune languages [21]; every $\leq_m^{\mathrm{P}}$-hard language for NP has an exponentially dense, exponentially hard complexity core [10]; and now, by Theorem 15, every $\leq_{n^\alpha-tt}^{\mathrm{P}}$-hard language for NP ($\alpha < 1$) is exponentially dense. Further investigation of the consequences and reasonableness of $\mu(\mathrm{NP}|\mathrm{E}_2) \neq 0$ and related strong, measure-theoretic hypotheses is clearly indicated.

Acknowledgments. The first author thanks Bill Gasarch and Alan Selman for helpful discussions. Both authors thank two anonymous referees for several suggestions that improved the exposition.

Note added in proof. Very recently, and independently, Fu [7] used resource-bounded Kolmogorov complexity to prove results similar to Theorem 15 but for Turing reducibilities instead of truth-table reducibilities.

REFERENCES

[1] N. Alon and J. H. Spencer, *The Probabilistic Method*, Wiley, New York, 1992.

[2] V. Arvind, J. Köbler, and M. Mundhenk, *Bounded truth-table and conjunctive reductions to sparse and tally sets*, Lecture Notes in Comput. Sci., 652, 1992, Springer-Verlag, Berlin, pp. 140–151.

[3] L. Berman and J. Hartmanis, *On isomorphism and density of* NP *and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[4] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. of Math. Stats., 23 (1952), pp. 493–509.

[5] P. Erdös, *Some remarks on the theory of graphs*, Bull. Amer. Math. Soc., 53 (1947), pp. 292–294.

[6] P. Erdös and J. Spencer, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[7] B. Fu, *With quasi-linear queries EXP is not polynomial time Turing reducible to sparse sets*, Proceedings of the Eighth Annual Structure in Complexity Theory Conference, San Diego, CA, IEEE Computer Society Press, New York, 1993, pp. 185–191.

[8] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1990), pp. 305–308.

[9] L. A. Hemachandra, M. Ogiwara, and O. Watanabe, *How hard are sparse sets?*, Proceedings of the Seventh Annual Structure in Complexity Theory Conference, Boston, MA, IEEE Computer Society Press, New York, 1992, pp. 222–238.

[10] D. W. Juedes and J. H. Lutz, *The complexity and distribution of hard problems*, SIAM J. Comput., to appear; also in Proceedings of the 34th IEEE Symposium on Foundations of Computer Science, Palo Alto, CA, IEEE Computer Society Press, New York, 1993, pp. 177–185.

[11] ———, *Kolmogorov complexity, complexity cores, and the distribution of hardness*, *Kolmogorov Complexity and Computational Complexity*, O. Watanabe, ed., Springer-Verlag, New York, 1992. pp. 43–65.

[12] R. M. Karp and R. J. Lipton, *Some connections between nonuniform and uniform complexity classes*, Proceedings of the 12th ACM Symposium on Theory of Computing, Los Angeles, CA, 1980, pp. 302–309; also published as *Turing machines that take advice*, Enseign. Math. 28 (1982), pp. 191–209.

[13] A. N. Kolmogorov and V. A. Uspenskii, *Algorithms and randomness*, Theory of Probab. Appl., 32 (1987), pp. 389–412.

[14] J. H. Lutz, *Intrinsically pseudorandom sequences*, in preparation.

[15] ———, *Resource-bounded measure*, in preparation.

[16] ———, *Category and measure in complexity classes*, SIAM J. Comput., 19 (1990), pp. 1100–1131.

[17] ———, *An upward measure separation theorem*, Theoret. Comput. Sci., 81 (1991), pp. 127–135.

[18] ———, *Almost everywhere high nonuniform complexity*, J. Comput. System Sci., 44 (1992), pp. 220–258.

[19] J. H. Lutz and W. J. Schmidt, *Circuit size relative to pseudorandom oracles*, Theoret. Comput. Sci. A, 107 (1993), pp. 95–120.

[20] S. R. Mahaney, *Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[21] E. MAYORDOMO, *Almost every set in exponential time is* P-*bi-immune*, Theoretical Computer Science, to appear; also in Seventeenth International Symposium on Mathematical Foundations of Computer Science, Prague, Czechoslovakia, Springer-Verlag, New York, 1992, pp. 392–400.

[22] A. R. MEYER, 1977, reported in [3].

[23] M. OGIWARA AND O. WATANABE, *On polynomial bounded truth-table reducibility of NP sets to sparse sets*, SIAM J. Comput., 20 (1991), pp. 471–483.

[24] A. L. SELMAN, personal communication, 1992.

[25] C. E. SHANNON, *A mathematical theory of communication*, Bell System Tech. J., 27 (1948), pp. 379–423, 623–656.

[26] ———, *The synthesis of two-terminal switching circuits*, Bell System Tech. J., 28 (1949), pp. 59–98.

[27] J. H. SPENCER, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

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

[29] V. A. USPENSKII, A. L. SEMENOV, AND A. KH. SHEN′, *Can an individual sequence of zeros and ones be random?*, Russian Math. Surveys, 45 (1990), pp. 121–189.

[30] O. WATANABE, *On the Structure of Intractable Complexity Classes*, Ph.D. thesis, Tokyo Institute of Technology, 1987.

[31] ———, *Polynomial time reducibility to a set of small density*, Proceedings of the Second Structure in Complexity Theory Conference, Ithaca, NY, 1987, pp. 138–146.

[32] C. B. WILSON, *Relativized circuit complexity*, J. Comput. System Sci., 31 (1985), pp. 169–181.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 780–788, August 1994

FINDING k DISJOINT PATHS IN A DIRECTED PLANAR GRAPH*

ALEXANDER SCHRIJVER†

Abstract. It is shown that, for each fixed k, the problem of finding k pairwise vertex-disjoint directed paths between given pairs of terminals in a directed planar graph is solvable in polynomial time.

Key words. disjoint, path, directed, planar, graph, polynomial-time, algorithm, free group, homologous, cohomologous

AMS subject classifications. 05C20, 05C85, 68R10

1. Introduction and statement of result. In this paper we show that the following problem, the *k disjoint paths problem for directed planar graphs*, is solvable in polynomial time, for any fixed k:

$$\text{(1)}\quad \begin{array}{rl} \text{given :} & \text{a directed planar graph } D = (V, A) \text{ and } k \text{ pairs } (r_1, s_1), \ldots, (r_k, s_k) \text{ of} \\ & \text{vertices of } D; \\ \text{find :} & k \text{ pairwise vertex-disjoint directed paths } P_1, \ldots, P_k \text{ in } D, \text{ where } P_i \text{ runs} \\ & \text{from } r_i \text{ to } s_i (i = 1, \ldots, k). \end{array}$$

The problem is NP-complete if we do not fix k (even in the undirected case; Lynch [2]). Moreover, it is NP-complete for $k = 2$ if we delete the planarity condition (Fortune, Hopcroft, and Wyllie [1]). This is in contrast to the undirected case (for those believing NP≠P), where Robertson and Seymour [4] showed that, for any fixed k, the k disjoint paths problem is polynomial-time solvable for any graph (not necessarily planar).

In this paper we do not aim at obtaining the best possible running time bound, as we presume that there are much faster (but possibly more complicated) methods for (1) than the one we describe in this paper. In fact, recently Reed, Robertson, Schrijver, and Seymour [3] showed that for undirected planar graphs the k disjoint paths problem can be solved in *linear* time, for any fixed k. This algorithm makes use of methods from Robertson and Seymour's theory of graph minors. A similar algorithm for directed planar graphs might exist but probably would require extending parts of graph minors theory to the directed case.

Our method is based on cohomology over free (nonabelian) groups. For the k disjoint paths problem we use free groups with k generators. It extends methods given in [5] for undirected graphs on surfaces based on homotopy. Cohomology is in a sense dual to homology and can be defined in any directed graph, even if it is not embedded on a surface. We apply cohomology to an *extension* of the planar graph dual of D—just applying homology to D itself seems not powerful enough.

We remark that in our approach free groups and (co)homology are used mainly as a framework to formulate certain ideas smoothly; they give us a convenient tool for recording shifts of curves over the plane. No deep group theory or topology is used. We could avoid free groups and cohomology by adopting a more complex notation and terminology; that would, however, implicitly mimic free groups and cohomology. The present approach also readily allows application of the algorithm where the embedding of the graph in the plane is given combinatorially, that is, by a list of the cycles that bound the faces of the graph.

*Received by the editors January 2, 1992; accepted for publication (in revised form) April 26, 1993.

†Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, the Netherlands and Department of Mathematics, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, the Netherlands.

2. The cohomology feasibility problem.

2.1. Free groups. The *free group* G_k, generated by the *generators* $g_1, g_2, \ldots, g_k$, consists of all words $b_1 b_2 \cdots b_t$ where $t \geq 0$ and $b_1, \ldots, b_t \in \{g_1, g_1^{-1}, \ldots, g_k, g_k^{-1}\}$ such that $b_i b_{i+1} \neq g_j g_j^{-1}$ and $b_i b_{i+1} \neq g_j^{-1} g_j$ for $i = 1, \ldots, t-1$ and $j = 1, \ldots, k$. The product $x \cdot y$ of two such words is obtained from the concatenation xy by deleting iteratively all occurrences of any $g_j g_j^{-1}$ and $g_j^{-1} g_j$. (So in our notation $x \cdot y \neq xy$ in general.) This defines a group, with unit element 1 equal to the empty word $\emptyset$. We call $g_1, g_1^{-1}, \ldots, g_k, g_k^{-1}$ the *symbols*. The *size* $|x|$ of a word x is the number of symbols occurring in it, counting multiplicities.

A word y is called a *segment* of a word w if $w = xyz$ for certain words x, z. It is called a *beginning segment* if $x = 1$ and an *end segment* if $z = 1$. A subset Γ of a free group is called *hereditary* if for each word $y \in \Gamma$, each segment of y belongs to Γ.

We define a partial order $\leq$ on G_k by

$$x \leq y \Leftrightarrow x \text{ is a beginning segment of } y. \tag{2}$$

This gives a lattice if we extend G_k with an element ∞ at infinity. We denote the meet and join by $\wedge$ and $\vee$. So $x \wedge y$ is equal to the longest common beginning segment of x and y. Moreover, $x \vee y = \infty$ except if $x \leq y$ or $y \leq x$.

We make two easy observations.

LEMMA 2.1. *Let α be a symbol, and let $x, z \in G_k$. If $x \leq \alpha \cdot z$ and $z \leq \alpha^{-1} \cdot x$, then $x^{-1} \cdot \alpha \cdot z = 1$ or $x = z = 1$.*

Proof. Let $y := x^{-1} \cdot \alpha \cdot z$, and suppose that $y \neq 1$. Since $x \leq \alpha \cdot z$, it follows that $\alpha \cdot z = xy'$ for some y', and hence $y = x^{-1} \cdot \alpha \cdot z = x^{-1} \cdot (xy') = y'$. Consequently $xy \in G_k$; and since $z \leq \alpha^{-1} \cdot x$, it follows similarly that $zy^{-1} \in G_k$, that is, $yz^{-1} \in G_k$. Since $y \neq 1$, this implies that $xyz^{-1} \in G_k$, and so $\alpha = x \cdot y \cdot z^{-1} = xyz^{-1}$. In particular, $1 = |\alpha| = |x| + |y| + |z| \geq |x| + 1 + |z|$. Therefore, $x = z = 1$. □

LEMMA 2.2. *Let $x, y \in G_k$. If $x \not\leq y$, then the last symbol of x is equal to the last symbol of $y^{-1} \cdot x$.*

Proof. Let $z := x \wedge y$. Write $x = zx'$ and $y = zy'$, where $x' \neq 1$. Let α be the first symbol of x'. Since $z = x \wedge y$, we know $\alpha \not\leq y'$. Hence $(y')^{-1} \cdot x' = (y')^{-1} x'$ (i.e., no cancellation). Consequently,

$$y^{-1} \cdot x = ((y')^{-1} z^{-1}) \cdot (zx') = (y')^{-1} \cdot x' = (y')^{-1} x'. \tag{3}$$

Hence, as $x' \neq 1$, the last symbol of x is equal to the last symbol of $y^{-1} \cdot x$. □

2.2. The cohomology feasibility problem for free groups. Let $D = (V, A)$ be a weakly connected directed graph, let $r \in V$, and let $(G, \cdot)$ be a group. (We allow directed graphs to have parallel arcs.) Two functions $\phi, \psi : A \longrightarrow G$ are called *r-cohomologous* if there exists a function $f : V \longrightarrow G$ such that

$$\begin{aligned} &\text{(i) } f(r) = 1; \\ &\text{(ii) } \psi(a) = f(u)^{-1} \cdot \phi(a) \cdot f(w) \text{ for each arc } a = (u, w). \end{aligned} \tag{4}$$

One easily checks that this gives an equivalence relation.

Consider the following *cohomology feasibility problem for free groups*:

$$\begin{aligned} &\text{given : a weakly connected directed graph } D = (V, A), \text{ a vertex } r, \text{ a function} \\ &\qquad \phi : A \longrightarrow G_k, \text{ and for each } a \in A \text{ a hereditary subset } \Gamma(a) of G_k; \\ &\text{find : a function } \psi : A \longrightarrow G_k \text{ such that } \psi \text{ is } r\text{-cohomologous to } \phi \text{ and such that} \\ &\qquad \psi(a) \in \Gamma(a) \text{ for each arc } a. \end{aligned} \tag{5}$$

We show the following.

THEOREM 2.3. *The cohomology feasibility problem for free groups is solvable in time bounded by a polynomial in* $|A| + \sigma + k$.

Here σ is the maximum size of the words $\phi(a)$ and the words in the $\Gamma(a)$. (In fact we can drop k and assume that G_k is the free group generated by the generators occurring in the $\phi(a)$ and the words in the $\Gamma(a)$.)

Note that, by the definition of r-cohomologous, equivalent to finding a ψ as in (5) is finding a function $f : V \longrightarrow G_k$ satisfying

$$(6) \quad \begin{array}{l} \text{(i) } f(r) = 1; \\ \text{(ii) } f(u)^{-1} \cdot \phi(a) \cdot f(w) \in \Gamma(a) \text{ for each arc } a = (u, w). \end{array}$$

We call such a function *f feasible.*

In solving the cohomology feasibility problem for free groups we may assume

$$(7) \quad \begin{array}{l} \text{(i) } \Gamma(a) \neq \emptyset \text{ for each arc } a; \\ \text{(ii) } |\phi(a)| \leq 1 \text{ for each arc } a; \\ \text{(iii) with each arc } a = (u, w) \text{ also } a^{-1} = (w, u) \text{ is an arc, with } \phi(a^{-1}) = \phi(a)^{-1} \\ \qquad \text{and } \Gamma(a^{-1}) = \Gamma(a)^{-1}. \end{array}$$

Here $\Gamma(a)^{-1} := \{x^{-1} | x \in \Gamma(a)\}$. Condition (7)(ii) can be attained by replacing any arc $a = (u, w)$ such that $\phi(a) = \beta_1 \cdots \beta_t$ and $t \geq 2$ by a $u - w$ path $a_1 \cdots a_t$ with $\phi(a_i) := \beta_i$ $(i = 1, \dots, t)$ and $\Gamma(a_1) := \Gamma(a)$ and $\Gamma(a_i) := \{1\}$ $(i = 2, \dots, t)$. (Here and below we indicate a path P by the string of arcs traversed by P (in the order traversed by P). If P traverses an arc a in the backward direction, then we denote this in the string by a^{-1}. For instance, $P = a_1 a_2^{-1} a_3$ means that P traverses first a_1 in the forward direction, next a_2 in the backward direction, and finally a_3 in the forward direction. The arcs need not be distinct.)

2.3. Pre-feasible functions. Let input $D = (V, A), r, \phi, \Gamma$ for the cohomology feasibility problem for free groups (5) be given, assuming (7). We call a function $f : V \longrightarrow G_k$ *pre-feasible* if

$$(8) \quad \begin{array}{l} \text{(i) } f(r) = 1; \\ \text{(ii) for each arc } a = (u, w) \text{ with } f(u)^{-1} \cdot \phi(a) \cdot f(w) \notin \Gamma(a) \text{ one has } f(u) = \\ \qquad f(w) = 1. \end{array}$$

Define a partial order $\leq$ on the set G_k^V of all functions $f : V \longrightarrow G_k$ by

$$(9) \qquad f \leq g \Leftrightarrow f(v) \leq g(v) \quad \text{for each } v \in V. \qquad \Box$$

It is easy to see that G_k^V forms a lattice if we add an element ∞ at infinity. Let $\wedge$ and $\vee$ denote the meet and join, respectively.

Pre-feasibility behaves nicely with respect to the lattice:

PROPOSITION 1. *If f_1 and f_2 are pre-feasible, then so is $f := f_1 \wedge f_2$.*

Proof. Clearly $f(r) = 1$. Let $a = (u, w)$ be an arc such that $y := f(u)^{-1} \cdot \phi(a) \cdot f(w) \notin \Gamma(a)$ while not $f(u) = f(w) = 1$. By Lemma 2.1 and by symmetry we may assume that $f(u) \not\leq \phi(a) \cdot f(w)$. Let x and x' be such that $f_1(u) = f(u)x$ and $f_2(u) = f(u)x'$, and let z and z' be such that $f_1(w) = f(w)z$ and $f_2(w) = f(w)z'$. Let α and β be the first and last symbol, respectively, of y. Since $z_1 \wedge z_2 = 1$, we know $\beta^{-1} \not\leq z$ or $\beta^{-1} \not\leq z'$. Without loss of generality, $\beta^{-1} \not\leq z$.

Since $f(u) \not\leq \phi(a) \cdot f(w)$, by Lemma 2.2 the first symbol of $f(u)^{-1}$ is equal to α. So $\alpha \not\leq x$, and hence

$$f_1(u)^{-1} \cdot \phi(a) \cdot f_1(w) = x^{-1} \cdot y \cdot z = x^{-1}yz. \tag{10}$$

So y is a segment of $f_1(u)^{-1} \cdot \phi(a) \cdot f_1(w)$. By the heredity of $\Gamma(a)$ this implies that $f_1(u)^{-1} \cdot \phi(a) \cdot f_1(w) \not\in \Gamma(a)$. So, as f_1 is pre-feasible, $f_1(u) = f_1(w) = 1$. Therefore $f(u) = f(w) = 1$. □

So for any function $f : V \longrightarrow G_k$ there exists a smallest pre-feasible function $\bar{f} \geq f$, provided there exists at least one pre-feasible function $g \geq f$. If no such g exists, we set $\bar{f} := \infty$. We observe the following proposition.

PROPOSITION 2. *If $\bar{f}$ is finite, then*

(i) $f(r) = 1$ *and* $|f(v)| < 2\sigma|V|$ *for each vertex* v;

(ii) *for each arc* $a = (u, w)$: *if* $f(u)^{-1} \cdot \phi(a) \cdot f(w) \not\in \Gamma(a)$, *then* $f(u) \leq \phi(a) \cdot f(w)$ *or* $f(w) \leq \phi(a^{-1}) \cdot f(u)$.

(11)

Proof. Clearly, $f(r) \leq \bar{f}(r) = 1$. Moreover, by induction on the minimum number t of arcs in any $r - v$ path one shows $|\bar{f}(v)| \leq 2\sigma t$. Indeed, if $a = (u, v)$ is the last arc in the path, then $y := \bar{f}(u)^{-1} \cdot \phi(a) \cdot \bar{f}(v)$ belongs to $\Gamma(a)$ or is equal to $\phi(a)$ and, hence, has size at most σ. So

$$|\bar{f}(v)| = |\bar{f}(u) \cdot \phi(a)^{-1} \cdot y| \leq |\bar{f}(u)| + |\phi(a)| + \sigma \leq 2\sigma(t-1) + 1 + \sigma \leq 2\sigma t. \tag{12}$$

This implies $|f(v)| \leq |\bar{f}(v)| < 2\sigma|V|$.

To see (ii), suppose $f(u) \not\leq \phi(a) \cdot f(w)$ and $f(w) \not\leq \phi(a^{-1}) \cdot f(u)$. The first implies (by Lemma 2.2) that the first symbol of $f(u)^{-1} \cdot \phi(a) \cdot f(w)$ is equal to the first symbol of $f(u)^{-1}$. The second implies (again by Lemma 2.2) that the last symbol of $f(u)^{-1} \cdot \phi(a) \cdot f(w)$ is equal to the last symbol of $f(w)$. Since $f \leq \bar{f}$, it follows that $f(u)^{-1} \cdot \phi(a) \cdot f(w)$ is a segment of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot \bar{f}(w)$. So $\bar{f}(u)^{-1} \cdot \phi(a) \cdot \bar{f}(w) \not\in \Gamma(a)$. Hence, as $\bar{f}$ is pre-feasible, $\bar{f}(u) = \bar{f}(w) = 1$, and therefore $f(u) = 1$. This contradicts the fact that $f(u) \not\leq \phi(a) \cdot f(w)$.
□

2.4. A subroutine finding $\bar{f}$. Let input $D = (V, A), r, \phi, \Gamma$ for the cohomology feasibility problem for free groups (5) be given, again assuming (7). We describe a polynomial-time subroutine that outputs $\bar{f}$ for any given $f : V \longrightarrow G_k$.

If f is pre-feasible, output $\bar{f} := f$. If f violates (11), output $\bar{f} := \infty$. Otherwise choose an arc $a = (u, w)$ satisfying $f(u)^{-1} \cdot \phi(a) \cdot f(w) \not\in \Gamma(a)$ and $f(w) \not\leq \phi(a^{-1}) \cdot f(u)$ (as f is not pre-feasible and satisfies (11), such an arc exists by Lemma 2.1). Perform the following:

Iteration: Write $\phi(a) \cdot f(w) = xy$, with $y \in \Gamma(a)$ and $|y|$ as large as possible, reset $f(u) := x$, and start anew.

PROPOSITION 3. *In the iteration, resetting f increases $|f(u)|$ and does not change $\bar{f}$.*

Proof. Consider the iteration. Denote by f' the reset f. As (11)(ii) holds, $f(u) \leq \phi(a) \cdot f(w)$. Since $f(u)^{-1} \cdot \phi(a) \cdot f(w) \not\in \Gamma(a)$, $f(u)$ is a segment of x with $f(u) \neq x$. So $|f'(u)| > |f(u)|$.

To see $\bar{f'} = \bar{f}$, we must show $f' \leq \bar{f}$, that is, $f'(u) \leq \bar{f}(u)$ if $\bar{f}$ is finite. Suppose that $\bar{f}$ is finite and that $f'(u) \not\leq \bar{f}(u)$. Let β be the last symbol of $x = f'(u)$. As $x \not\leq \bar{f}(u)$ and as $\phi(a) \cdot f(w) = xy$, βy is an end segment of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot f(w)$.

Since $\phi(a) \cdot f(w) \not\leq \bar{f}(u)$ (as $x \leq \phi(a) \cdot f(w)$), by Lemma 2.2 the last symbol of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot f(w)$ is equal to the last symbol of $\phi(a) \cdot f(w)$. Since $\phi(a) \cdot f(w) \not\leq f(u)$ (as $x \leq \phi(a) \cdot f(w)$ and $f(u) \leq \bar{f}(u)$), by Lemma 2.2 the last symbol of $\phi(a) \cdot f(w)$ is equal to the last symbol of $f(u)^{-1} \cdot \phi(a) \cdot f(w)$. Since $f(w) \not\leq \phi(a^{-1}) \cdot f(u)$, by Lemma 2.2 the last symbol of $f(u)^{-1} \cdot \phi(a) \cdot f(w)$ is equal to the last symbol of $f(w)$. Concluding, the last symbol of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot f(w)$ is equal to the last symbol of $f(w)$. Hence $\bar{f}(u)^{-1} \cdot \phi(a) \cdot f(w)$ is a beginning segment of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot \bar{f}(w)$. So βy is a segment of $\bar{f}(u)^{-1} \cdot \phi(a) \cdot \bar{f}(w)$, and hence βy belongs to $\Gamma(a)$. This contradicts the maximality of y. □

Since at each iteration $|f(u)|$ increases for some vertex u, after at most $2\sigma|V|^2$ iterations either we get a prefeasible function f or (11) is violated. Thus the subroutine is polynomial time.

2.5. Algorithm for the cohomology feasibility problem for free groups. Let input $D = (V, A), r, \phi, \Gamma$ for the cohomology feasibility problem for free groups (5) be given. We find a feasible function f as follows.

Again we may assume (7). For every $a = (u, w) \in A$ let f_a be the function defined by $f_a(u) := \phi(a)$ and $f_a(v) := 1$ for each $v \neq u$. Let E be the set of pairs $\{a, a'\}$ from A for which $\bar{f}_a \vee \bar{f}_{a'}$ is finite and pre-feasible. Let E' be the set of pairs $\{a, a^{-1}\}$ with $a \in A$ and $\phi(a) \notin \Gamma(a)$.

We search for a subset X of A such that each pair in X belongs to E and such that X intersects each pair in E'. This is a special case of the two-satisfiability problem and, hence, can be solved in polynomial time.

PROPOSITION 4. *If X exists, then the function $f := \bigvee_{a \in X} \bar{f}_a$ is feasible. If X does not exist, then there is no feasible function.*

Proof. First assume that X exists. Let f be as given. Since $\bar{f}_a \vee \bar{f}_{a'}$ is finite and pre-feasible for each two a, a' in X, f is finite and $f(r) = 1$. Moreover, suppose $f(u)^{-1} \cdot \phi(a) \cdot f(w) \notin \Gamma(a)$ for some arc $a = (u, w)$. Let $f(u) = \bar{f}_{a'}(u)$ and $f(w) = \bar{f}_{a''}(w)$ for $a', a'' \in X$. As $\bar{f}_{a'} \vee \bar{f}_{a''}$ is pre-feasible, $\bar{f}_{a'}(u) = \bar{f}_{a''}(w) = 1$. So $\phi(a) \notin \Gamma(a)$, and hence a or a^{-1} belongs to X. By symmetry we may assume $a \in X$. Then

$$\phi(a) = f_a(u) \leq \bar{f}_a(u) \leq f(u) = \bar{f}_{a'}(u) = 1, \tag{13}$$

a contradiction.

Assume conversely that there exists a feasible function f. Let X be the set of arcs $a = (u, w)$ with the property that $\phi(a) \leq f(u)$. Then X intersects each pair in E'. For let $a = (u, w)$ be an arc satisfying $a \notin X$ and $a^{-1} \notin X$, that is, $\phi(a) \not\leq f(u)$ and $\phi(a^{-1}) \not\leq f(w)$. Hence $f(u)^{-1} \cdot \phi(a) \cdot f(w) = f(u)^{-1}\phi(a)f(w)$. So $f(u)^{-1} \cdot \phi(a) \cdot f(w)$ contains $\phi(a)$ as a segment (as $|\phi(a)| \leq 1$). So $\phi(a) \in \Gamma(a)$ and hence $\{a, a^{-1}\} \notin E'$.

Moreover, each pair in X belongs to E. For let $\{a', a''\}$ be a pair in X. We show that $\{a', a''\} \in E$, that is, $f' := \bar{f}_{a'} \vee \bar{f}_{a''}$ is finite and pre-feasible. As $a' \in X$, we have $\phi(a') \leq f(u)$ and hence $f_{a'} \leq f$, implying $\bar{f}_{a'} \leq f$. Similarly, $\bar{f}_{a''} \leq f$. So f' is finite and $f'(r) = 1$. Consider an arc $a = (u, w)$ with $y := f'(u)^{-1} \cdot \phi(a) \cdot f'(w) \notin \Gamma(a)$. We may assume $f'(u) = \bar{f}_{a'}(u)$ and $f'(w) = \bar{f}_{a''}(w)$ (since $\bar{f}_{a'}$ and $\bar{f}_{a''}$ are pre-feasible). To show $f'(u) = f'(w) = 1$, by Lemma 2.1 we may assume $f'(w) \not\leq \phi(a^{-1}) \cdot f'(u)$.

First assume $f'(u) \not\leq \phi(a) \cdot f'(w)$. Then by Lemma 2.2 the first and the last symbols of y are equal to the first symbol of $f'(u)^{-1}$ and the last symbol of $f'(w)$, respectively. Since $f' \leq f$, this implies that y is a segment of $f(u)^{-1} \cdot \phi(a) \cdot f(w) \in \Gamma(a)$. This contradicts the heredity of $\Gamma(a)$ as $y \notin \Gamma(a)$.

Second assume $f'(u) \leq \phi(a) \cdot f'(w)$. So $\phi(a) \cdot f'(w) = f'(u)y$ for some y. Since $\bar{f}_{a''}(u) \leq f'(u)$, it follows that y is an end segment of

$$(14) \qquad \bar{f}_{a''}(u)^{-1} \cdot (f'(u)y) = \bar{f}_{a''}(u)^{-1} \cdot \phi(a) \cdot f'(w) = \bar{f}_{a''}(u)^{-1} \cdot \phi(a) \cdot \bar{f}_{a''}(w).$$

So $\bar{f}_{a''}(u)^{-1} \cdot \phi(a) \cdot \bar{f}_{a''}(w) \notin \Gamma(a)$, since $y \notin \Gamma(a)$. As $\bar{f}_{a''}$ is pre-feasible, this implies $\bar{f}_{a''}(u) = \bar{f}_{a''}(w) = 1$; so $f'(w) = 1$. Hence $f'(u) \leq \phi(a)$ and therefore, since $y \notin \Gamma(a)$ and $|\phi(a)| \lesssim 1$, $f'(u) = 1$. □

Thus we have proved Theorem 2.3.

3. The k disjoint paths problem for directed planar graphs.

3.1. Directed planar graphs, R-homology, and flows. Let input $D = (V, A)$, $r_1, s_1, \ldots, r_k, s_k$ for problem (1) be given. In solving (1) we may assume that D is weakly connected and that for each $i = 1, \ldots, k$, r_i is incident with exactly one arc, which is leaving r_i, and s_i is incident with exactly one arc, which is entering s_i. We fix an embedding of D. Let $\mathcal{F}$ denote the collection of faces of D and let R be the unbounded face of D.

Call two functions $\phi, \psi : A \longrightarrow G_k$ *R-homologous* if there exists a function $f : \mathcal{F} \longrightarrow G_k$ such that

(i) $f(R) = 1$;

(15) (ii) $f(F)^{-1} \cdot \phi(a) \cdot f(F') = \psi(a)$ for each arc a, where F and F' are the faces at the left-hand side and right-hand side of a, respectively.

The relation to cohomologous is direct by duality. The *dual* graph $D^* = (\mathcal{F}, A^*)$ of D has as vertex set the collection $\mathcal{F}$ of faces of D, while for any arc a of D there is an arc a^* of D^* from the face of D at the left-hand side of a to the face at the right-hand side. Define for any function ϕ on A the function ϕ^* on A^* by $\phi^*(a^*) := \phi(a)$ for each $a \in A$. Then ϕ and ψ are R-homologous (in D) if and only if ϕ^* and ψ^* are R-cohomologous (in D^*).

For any solution $\Pi = (P_1, \ldots, P_k)$ of (1) let $\psi_\Pi : A \longrightarrow G_k$ be defined by $\psi_\Pi(a) := g_i$ if path P_i traverses a ($i = 1, \ldots, k$), and $\psi_\Pi(a) := 1$ if a is not traversed by any of the P_i.

Call a function $\phi : A \longrightarrow G_k$ a *flow* if for each vertex $v \in V$ with $v \notin \{r_1, s_1, \ldots, r_k, s_k\}$ one has

$$(16) \qquad \phi(a_1)^{\varepsilon_1} \cdot \phi(a_2)^{\varepsilon_2} \cdot \cdots \cdot \phi(a_n)^{\varepsilon_n} = 1,$$

where $a_1, \ldots, a_n$ are the arcs incident with v, in clockwise order, where $\varepsilon_i = +1$ if a_i has its tail at v and $\varepsilon_i = -1$ if a_i has its head at v (if a_i happens to be a loop we take $\varepsilon_i = +1$ and $\varepsilon_i = -1$ at the corresponding positions in (16)), and if moreover for any arc a incident with r_i or s_i one has $\phi(a) = g_i$ ($i = 1, \ldots, k$).

Clearly, if Π is a solution of (1), then ψ_Π is a flow. Moreover, if ϕ is a flow and ϕ' is R-homologous to ϕ, then also ϕ' is a flow.

3.2. Deriving a solution from a flow. We first show the next proposition.

PROPOSITION 5. *There exists a polynomial-time algorithm that, for any flow ϕ, either finds a solution Π of (1) or concludes that there does not exist a solution Π of (1) such that ψ_Π is R-homologous to ϕ.*

[Here *polynomial-time* means: polynomial-time in the size of D and the maximum size of the $\phi(a)$. Note that it is not required that if we find a solution Π of (1), then ψ_Π is R-homologous to ϕ.]

Proof. Let ϕ be a flow. Consider the dual graph $D^* = (\mathcal{F}, A^*)$ of D. We construct the 'extended' dual graph $D^+ = (\mathcal{F}, A^+)$ by adding in each face of D^* all chords. (So D^+ need not be planar.) More precisely, for any two vertices F, F' of D^* and any (undirected) $F - F'$

path π on the boundary of any face of D^*, extend the graph with an arc, denoted by a_π, from F to F'. Define $\phi^+ : A^+ \longrightarrow G_k$ by

(17)
(i) $\phi^+(a^*) := \phi(a)$ for each arc a of D;
(ii) $\phi^+(a_\pi) := \phi(a_1)^{\varepsilon_1} \cdot \dots \cdot \phi(a_t)^{\varepsilon_t}$ for any path $\pi = (a_1^*)^{\varepsilon_1} \cdots (a_t^*)^{\varepsilon_t}$ with $\varepsilon_1, \dots, \varepsilon_t \in \{+1, -1\}$.

(As before, $(a_i^*)^{-1}$ means that π traverses a_i^* in the backward direction.) Moreover, let $\Gamma(a^*) := \{1, g_1, \dots, g_k\}$ and $\Gamma(a_\pi) := \{1, g_1, g_1^{-1}, \dots, g_k, g_k^{-1}\}$.

By Theorem 2.3 we can find in polynomial time a function ψ that is R-cohomologous to ϕ^+ in D^+, with $\psi(b) \in \Gamma(b)$ for each arc b of D^+, provided that such a ψ exists. If we find such a ψ, let P_i be any directed $r_i - s_i$ path traversing only arcs a satisfying $\psi(a^*) = g_i$ (for $i = 1, \dots, k$). (Such paths exist since ϕ is a flow.) Then $P_1, \dots, P_k$ form a solution to (1). Indeed, $P_1, \dots, P_k$ are vertex-disjoint, for suppose that there exist arcs a and b of D that are both incident with a vertex v and $\psi(a^*) = g_i^{\pm 1}$, $\psi(b^*) = g_j^{\pm 1}$, and $i \neq j$. Consider a shortest path π along the face of D^* corresponding to v such that π contains arcs a^* and b^*. We may assume that we have chosen a and b such that π is as short as possible. Then $|\psi(a_\pi)| \geq 2$, as $\psi(a_\pi)$ contains both $g_i^{\pm 1}$ and $g_j^{\pm 1}$ (neither of them can be cancelled, since a and b are chosen so that π is shortest). This contradicts the fact that $\psi(a_\pi) \in \Gamma(a_\pi)$.

If we do not find such a function ψ, we may conclude that there does not exist a solution Π of (1) such that ψ_Π is R-homologous to ϕ, since otherwise the cohomology feasibility problem has a solution, viz. $\psi := (\psi_\Pi)^+$. □

3.3. Enumerating homology types. In this section we show the following.

PROPOSITION 6. *For each fixed k, we can find in polynomial time flows $\phi_1, \dots, \phi_N$ with the property that for each solution Π of* (1), *ψ_Π is R-homologous to at least one of $\phi_1, \dots, \phi_N$.*

Proof. Consider systems $\Pi = (P_1, \dots, P_k)$ satisfying:

(18)
(i) P_i is an undirected path from r_i to s_i, not traversing the same edge more than once, and not having any self-crossing ($i = 1, \dots, k$);
(ii) P_i and P_j are edge-disjoint and do not have any crossing ($i, j = 1, \dots, k$; $i \neq j$).

(An *undirected* path is a path that may traverse arcs in the backward direction.)

For any such system Π, define $\psi_\Pi : A \longrightarrow G_k$ by $\psi_\Pi(a) := g_i$ if P_i traverses a in the forward direction, $\psi_\Pi(a) := g_i^{-1}$ if P_i traverses a in the backward direction ($i = 1, \dots, k$), and $\psi_\Pi(a) := 1$ if a is not traversed by any P_i.

We will show that for each fixed k, we can find in polynomial time flows $\phi_1, \dots, \phi_N$ with the property that

(19) for each Π satisfying (18), ψ_Π is R-homologous to at least one of $\phi_1, \dots, \phi_N$.

This is stronger than what we need to show.

Consider a nonloop arc a' not incident with any r_i or s_i. Contract a', yielding graph D'. Let $\phi_1', \dots, \phi_N'$ be flows in D' satisfying (19) with respect to D'. Then for each j there is a unique flow ϕ_j in D such that $\phi_j(a) = \phi_j'(a)$ for each arc $a \neq a'$. Moreover, if Π satisfies (18) in D, then contracting a' gives a system Π' satisfying (18) in D'. Hence there exists a j such that ϕ_j' is R-homologous to $\psi_{\Pi'}$ (in D'), implying that ϕ_j is R-homologous to ψ_Π (in D).

Concluding, we can obtain from a system of flows satisfying (19) for D' a system of flows satisfying (19) for D. Repeating this we obtain that we may assume that there is only one vertex v in $V \setminus \{r_1, s_1, \dots, r_k, s_k\}$ and that each arc not incident with $r_1, s_1, \dots, r_k, s_k$ is a loop at v. We may assume that each loop is oriented clockwise (since presently we are interested in undirected paths). For each loop l let X_l be the set of vertices in $r_1, s_1, \dots, r_k, s_k$ enclosed by l. Call loops l, l' *parallel* if $X_l = X_{l'}$. Trivially, there are at most 2^{2k} parallel classes. (By Euler's theorem, there are at most $4k$ parallel classes, but we do not need this stronger bound, since k is fixed.)

If Π satisfies (18), then there is a system Π' satisfying (18) such that $\psi_{\Pi'}$ is R-homologous to ψ_Π and such that the paths in Π do not contain any loop l with $X_l = \emptyset$ and do not contain $l'l^{-1}$ or $l^{-1}l'$ for any two parallel loops l, l'. So we can restrict the systems Π to systems having this additional property.

For any such system Π and any two subsets $B, C \subseteq \{a, a^{-1} | a \in A\}$, let $x_\Pi(B, C)$ denote the number of occurrences of bc in the paths in Π such that $b \in B, c \in C$. Then Π is up to R-homology fully determined by the system of numbers $x_\Pi(B, C)$, where B and C range over all sets

$$(20) \qquad \begin{array}{ll} L, L^{-1} & (L \text{ a parallel class of loops}), \\ \{(r_i, v)\}, \{v, s_i\} & (i = 1, \dots, k), \end{array}$$

with $L^{-1} := \{l^{-1} | l \in L\}$. Since each such number $x_\Pi(B, C)$ is at most $|A|$ and since there are at most $2(k+2^{2k})$ sets among (20), for fixed k we can enumerate all possibilities in polynomial time. □

3.4. The disjoint paths problem.

THEOREM 3.1. *For each fixed k, the k disjoint paths problem for directed planar graphs* (1) *is solvable in polynomial time.*

Proof. By Proposition 3.3 we can find in polynomial time (fixing k) a list of flows $\phi_1, \dots, \phi_N$ such that for each solution Π of (1), ψ_Π is R-homologous to at least one of the ϕ_j.

Now for each $j = 1, \dots, N$ we apply the algorithm of Proposition 3.2 to input ϕ_j. If for some j we find a solution Π of problem (1), we are done. If for each of $j = 1, \dots, N$ it concludes that there is no solution Π of (1) such that ψ_Π is R-homologous to ϕ_j, we may conclude that (1) has no solution at all. □

Quite directly one can extend the method to the following problem:

(21)
given : a directed planar graph $D = (V, A)$, k pairs $(r_1, s_1), \dots, (r_k, s_k)$ of vertices
of D, and subsets $A_1, \dots, A_k$ of A;
find : k pairwise vertex-disjoint directed paths $P_1, \dots, P_k$ in D, where P_i runs
from r_i to s_i and uses only arcs in $A_i (i = 1, \dots, k)$.

The polynomial-time solvability of this problem (for fixed k) follows by restricting in the proof of Proposition 3.2 each $\Gamma(a^*)$ to those g_i for which A_i contains a.

Acknowledgments. I am grateful to Coelho de Pina, Paul Seymour, and an anonymous referee for carefully reading the manuscript, for pointing out some inaccuracies, and for giving several helpful suggestions improving the presentation.

REFERENCES

[1] S. FORTUNE, J. HOPCROFT, AND J. WYLLIE, *The directed subgraph homeomorphism problem*, Theoret. Comput. Sci., 10 (1980), pp. 111–121.

[2] J. F. Lynch, *The equivalence of theorem proving and the interconnection problem*, (ACM) SIGDA Newsletter 5, 3 (1975), pp. 31–36.

[3] B. Reed, N. Robertson, A. Schrijver, and P. D. Seymour, *Finding disjoint trees in planar graphs in linear time*, in Graph Structure Theory, N. Robertson and P. D. Seymour, eds., Contemp. Math. 147, Amer. Math. Soc., Providence, RI, 1993, pp. 295–301.

[4] N. Robertson and P. D. Seymour, *Graph minors* XIII. *The disjoint paths problem*, preprint, 1986.

[5] A. Schrijver, *Disjoint homotopic paths and trees in a planar graph*, Discrete Comput. Geom., 6 (1991), pp. 527–574.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 789–814, August 1994

FINITE AUTOMATA COMPUTING REAL FUNCTIONS*

KAREL CULIK II† AND JUHANI KARHUMÄKI‡

Abstract. A new application of finite automata as computers of real functions is introduced. It is shown that even automata with a restricted structure compute all polynomials, many fractal-like and other functions. Among the results shown, the authors give necessary and sufficient conditions for continuity, show that continuity and equivalence are decidable properties, and show how to compute integrals of functions in the automata representation.

Key words. weighted finite automata, fractals, image generation, data compression

AMS subject classifications. 68Q45, 68Q99, 68U05, 68Q10

1. Introduction. We study weighted finite automata that are finite automata with transitions (edges in a diagram) labeled by real numbers. Such automata have already been extensively studied with various motivations; see, for example, [10] for a general definition and theoretical properties.

Historically, such devices were originally studied as probabilistic automata [12], first mainly as language acceptors like ordinary automata and later as devices defining a probability distribution on Σ^* with various application, for example, in learning theory [14]. As a quite different approach, attempts to generalize and at the same time to unify results of classical automata theory lead to the study of formal power series [13] that was based on weighted finite automata.

Recently, a new and different application of weighted finite automata was introduced in [3], where they were called probabilistic finite generators. They were used as a formal specification of grey-scale images as an alternative to the more analytical approach of [1]. Our theoretical work is motivated by these practical applications. The usefulness of this approach is demonstrated in [3], [5], [4]. Recent work [6] indicates that weighted finite automata are an excellent tool for data compression, especially in combination with wavelets.

We will concentrate mainly on weighted automata with special structures called level automata and line automata. However, even this special subset of weighted automata is powerful enough to generate all polynomials and many other interesting functions including many fractal-type functions.

We give the basic definitions and properties in §2. Our basic results, including the necessary and sufficient condition for a function $\hat{f}_A$ defined by level automaton A to be continuous, are shown in §3. This result is then used to show that it is decidable whether a given level automaton defines a continuous function. Further, we show that the problem whether two level automata define the same real function is decidable.

In §4 we study in detail the line automata of degree one that are weighted automata with two states that are interconnected in one direction only. Our main observation here is that the only smooth functions defined by the line automata of degree one are the constant and the linear functions but that such automata also define continuous fractal-type functions, that is, functions with self-similar graphs, including for example a function with a dense (or even uncountable) set of points where it does not have a derivative, and also functions that are not continuous.

*Received by the editors December 13, 1991; accepted for publication (in revised form) May 12, 1993. This work was performed during the first author's stay at the University of Turku, Finland and was supported by the Academy of Finland and partially supported by the National Sciences Foundation grant CCR-9202396.

†Department of Computer Science, University of South Carolina, Columbia, South Carolina 29208 (culik@cs.scarolina.edu).

‡Department of Mathematics, University of Turku, Turku, 20500 Finland.

Our main and surprising result in §5 is that all polynomials of one variable and degree up to n are defined by line automata with the same structure and weights (only the initial distribution determines a particular polynomial). For polynomials of more variables an analogous result holds for level automata that are weighted automata having only loops of length one.

In §6 we consider average preserving weighted automata that are such automata that for each node the sum of all weights at all transitions is equal to 2^d where d is the dimension, i.e., the space $[0, 1]^d$ is considered. We use the fact, shown in §3, that every weighted level automaton can be converted into an average preserving level automaton defining the same real function. Then we show a rather surprising result that for every level automaton A in one dimension we can easily construct a level automaton B such that $\hat{f}_B(x) = \int_0^x \hat{f}_A(t)\,dt$ for each $x \in [0, 1]$. For automata defining smooth functions we cannot only integrate but also mechanically compute derivatives.

2. Preliminaries. Let Σ be a finite alphabet and $\Sigma^\star$ the monoid generated by Σ. Elements of $\Sigma^\star$ are called words; in particular, the unit of $\Sigma^\star$ is called the empty word and denoted by ϵ. For a word u we denote its length by $|u|$, its prefix of length k by $\mathrm{pref}_k(u)$, and the number of letters a in u by $|u|_a$.

In this paper we are mainly concerned with (one-way) infinite words or ω-words over Σ. Formally, they can be viewed as mappings $\mathbb{N} \to \Sigma$ or, equivalently, as sequences $a_1a_2\ldots$ with $a_i \in \Sigma$. The set of all infinite words over Σ is denoted by Σ^ω. Clearly, the operator pref_k extends to Σ^ω as well. If there is no danger of confusion, we call infinite words simply words.

As is well known, each infinite word can be interpreted as a real number in the interval $[0, 1]$. We make use of this interpretation. When doing so we assume throughout this paper, if not stated otherwise, that Σ is binary, say $\Sigma = \{0, 1\}$. Hence, the infinite word

$$w = a_1a_2\ldots \quad \text{with } a_i \in \{0, 1\} \tag{1}$$

is interpreted as the number

$$\hat{w} = \sum_{i=1}^{\infty} a_i 2^{-i}. \tag{2}$$

Clearly, the only real numbers in $[0, 1]$ that have two representations (1) are those that have a finite representation (2), i.e., the numbers that have a finite binary representation. For such words the two representations are $v10^\omega$ and $v01^\omega$ for some finite v. We refer to the first one as the *standard* representation of the considered number and may write instead of $v10^\omega$ simply $v1$. It follows that the mapping $\wedge$:

$$\wedge : \Sigma^\omega \to [0, 1],\ \wedge(w) = \hat{w}$$

is one-to-one if and only if we restrict it to $\Sigma^\omega - \Sigma^\star 1^\omega$.

We also want to use infinite words to represent d-dimensional vectors in $[0, 1]^d$. Then we proceed as follows. First we present the ith component of the vector $x = (x_1, \ldots, x_d)$ as the binary ω-word $w_i \in \Sigma^\omega$. Then we make use of the inclusion

$$(\Sigma^d)^\omega \subseteq (\Sigma^\omega)^d$$

and say that x is represented by the ω-word

$$b_1b_2\ldots \quad \text{with } b_j = (a_{1j}, \ldots, a_{dj}),$$

where a_{ij} is the jth symbol of w_i. Consequently, the real vectors in $[0, 1]^d$ are represented by ω-words over the alphabet of size 2^d and not by d tuples of binary ω-words. From the point of view of automata theory this is a clear advantage. The notion of the standard representation obviously extends from numbers to vectors.

We now introduce our basic tool to define real functions, namely, that of a weighted finite automaton. A *weightedfinite automaton*, (WFA, for short) is a five-tuple $A = (Q, \Sigma, W, I, T)$, where

- Q is a finite set of *states* of the cardinality t,
- Σ is a finite *alphabet*,
- $W : Q \times \Sigma \times Q \to \mathbb{R}$ is a *weight function*,
- $I : Q \to \mathbb{R}$ is the *initial distribution*,
- $T : Q \to \mathbb{R}$ is the *final distribution*.

Clearly, the weight function W decomposes in a natural way to functions W_a for $a \in \Sigma$ and each W_a can be viewed as a $t \times t$ matrix over reals. We use the same notation W_a for these matrices. Consequently, if $W(p, a, q) = \alpha$, then the matrix W_a contains α at the entry of the row corresponding to the state p and of the column corresponding to the state q. As we shall see it is natural to view I as a row vector and T as a column vector. Note that weighted finite automata are called $(\mathbb{R} - \Sigma)$-automata in [10]. By the *underlying automaton* of a WFA A we mean the nondeterministic automaton whose transitions consist of those triples (p, a, q) for which $W(p, a, q) \neq 0$, and initial and final states are those that get nonzero value under I and T, respectively. Hence, we can talk about initial and final states of a WFA as well. This notion is used in the definition of our central notion of a level automaton.

As is well known, weighted finite automata can be used to compute functions $\Sigma^\star \to \mathbb{R}$, so-called $\mathbb{R}$-*rational* functions, as follows; cf. [10], [2], [13]. First, the *distribution* defined by $u \in \Sigma^\star$ on A, in symbols $P_A(u)$, is defined recursively:

$$\begin{aligned} P_A(\varepsilon) &= I, \\ P_A(va) &= P_A(v) \cdot W_a, \quad \text{for } v \in \Sigma^\star. \end{aligned}$$

Then the rational function determined by A, in symbols F_A, is defined by

$$F_A(u) = P_A(u) \cdot T.$$

Of course, the product here is the matrix product.

We want to use WFA to compute real functions on the interval $[0, 1]$ (or on the cube $[0, 1]^d$). This is achieved by extending the above word functions to infinite words corresponding to reals or to real vectors. Consequently, we now assume that $\Sigma = \{0, 1\}$ (or $\{0, 1\}^d$, possibly renamed). Then we say that A defines the *ω-word function*

$$(3) \qquad f_A : \Sigma^\omega \to \mathbb{R}, \qquad f_A(w) = \left(\lim_{n\to\infty} P_A(\mathrm{pref}_n(w))\right) \cdot T$$

and the *real function*

$$(4) \qquad \hat{f}_A : [0, 1] \to \mathbb{R}, \qquad \hat{f}_A(x) = f_A(w),$$

where w is the standard representation of x.

The definitions of f_A and $\hat{f}_A$ tacitly assume the existence of the limit in (3). We do not analyze this in detail here but instead study several special types of automata, which make these definitions mathematically sound (cf. §3) and at the same time are suitable for our results.

We call a WFA A a *level automaton* if and only if the following conditions are satisfied:

(i) The only loops in the underlying automata of A are transitions of the form $p \xrightarrow{a} p$.

(ii) $W(p, a, q) \geq 0$ for all $p, q \in Q, a \in \Sigma$.

(iii) For every $p \in Q$ the following holds. If there are $q \in Q,\ a \in \Sigma,\ q \neq p$ such that $W(p, a, q) \neq 0$ for some a, then $W(p, a, q) < 1$ for every $a \in \Sigma$ and $q \in Q$. If there are no such q and a, then $W(p, a, p) = 1$ for every $a \in \Sigma$.

(iv) $I \in \mathbb{R}_+^t$ and $T = \mathbb{R}_+^t$.

(v) The underlying automaton of A is reduced that is does not have useless states.

By condition (i) we can associate with each state of a WFA A a number called its *degree*, as follows:

- A state that leads in the underlying automaton only to itself is of degree 0.
- A state that is not of a degree smaller than i and leads in the underlying automaton only to states of degrees at most $i - 1$ or itself is of degree i.

Finally, a level automaton A is a *line* automaton if and only if for each number $0, \dots, \mathrm{card}(Q) - 1$ there exists exactly one state of this degree.

Several comments concerning the choice of conditions (i)–(v) are in order. Condition (i) strongly restricts the family of considered WFA. This, however, is needed in order to be able to handle WFA and, as we shall see, still gives a class large enough to yield interesting and surprising results. Conditions (ii) and (iv) mean that the functions F_A are $\mathbb{R}_+$-rational and not only $\mathbb{R}$-rational. The latter are more general than the former; cf. [2], [10], [13]. After (ii) and (iv) condition (iii) is natural: it guarantees that $\hat{f}_A$ is bounded, cf. Lemma 6. Finally, condition (v) is a standard one in automata theory. Concerning the choice of I and T we make the following conventions: *If not otherwise stated, we assume that* $I = (1, 0, \dots, 0)$ *and* $T = (1, 1, \dots, 1)$; however, as we shall see in the proof of Lemma 6 in a level automaton the final distribution $(1, 1, \dots, 1)$ can always be replaced by $(0, \dots, 0, 1)$ without affecting the function the automaton computes. In §5 we allow negative values of I in order to synthesize polynomials with negative coefficients easily.

To conclude we define two particular classes of WFA. We say that WFA A over $\{0, 1\}$ is 0-*faithful* (respectively 1-*faithful*) if and only if

$$\sum_{q \in Q} W(p, 0, q) = 1 \quad \text{for all } p \text{ in } Q$$

$$\left(\text{respectively} \quad \sum_{q \in Q} W(p, 1, q) = 1 \quad \text{for all } p \text{ in } Q\right).$$

Further, a WFA A over Σ^d, with $\Sigma = \{0, 1\}$, is *average preserving* if and only if

$$\sum_{q \in Q, a \in \Sigma} W(p, a, q) = 2^d \quad \text{for all } p \text{ in } Q.$$

These special classes of WFA have nice properties assuming that we have our standard choice $T = (1, 1, \dots, 1)$.

LEMMA 1. *If a* WFA A *is* 0-*faithful, then*

$$F_A(v) = f_A(v0^\omega) \quad \text{for all } v \text{ in } \Sigma^\star.$$

In other words, if $x \in [0, 1)$ *has a finite binary representation* v, *then* $F_A(v) = \hat{f}_A(x)$.

Proof. Straightforward by definitions. □

A similar result holds for 1-faithful WFA as well.

LEMMA 2. *If a* WFA A *is average preserving, then*

$$F_A(v) = \frac{1}{2^d} \sum_{a \in \Sigma^d} F_A(va) \quad \text{for all } v \in \Sigma^\star. \tag{5}$$

Proof. Again straightforward by definitions. □

Functions satisfying the condition of Lemma 2 are called *average preserving* functions (ap-functions, for short). The name is self-explanatory: its value at any point v (read: any pixel) is the average of its values on points va (read: its subpixels).

Average preserving functions $\Sigma^\star \to \mathbb{R}_+$ are closely related to *measures*, (cf. [1]), defined on Borel subsets of the complete metric space $(\Sigma^d)^\omega$ or $[0, 1]^d$, which are additive functions mapping the whole space X to 1, i.e., $\mu : (\Sigma^d)^\omega \to \mathbb{R}_+$ is a measure (called a normalized measure in [1]) if for each countable I and pairwise disjoint collection $\{A_i\}_{i\in I}$ of Borel subsets of $(\Sigma^d)^\omega$

$$\mu\left(\bigcup_{i\in I} A_i\right) = \sum_{i\in I} \mu(A_i) \tag{6}$$

and

$$\mu((\Sigma^d)^\omega) = 1.$$

Consequently, a measure function gives a weight (or greyness) for each (Borel) subset of X.

A WFA A satisfying

$$\sum_{p,q\in Q, a\in\Sigma} W(p,a,q) = 1,$$

defines a measure function μ_A on (Borel) subsets of $(\Sigma^d)^\omega$. Indeed, the automaton starting with the distribution $(1, 0, \ldots, 0)$ redistributes in an additive way the total amount of ink (equal to 1) to different pixels; the pixel addressed by the word v gets $F_A(v)$ amount of ink.

The connection between average preserving WFA and the measure defining WFA is straightforward due to (5) and (6). If A is an average preserving WFA, then by dividing each of its weights by 2^d we obtain a WFA that defines a measure function. The converse transformation is equally easy: weights must be multiplied by 2^d. However, this transformation does not necessarily preserve level automata; therefore, the limit in (3) might not exist and $\hat{f}(x)$ might be undefined.

3. Basic results. In this section we prove several basic results on WFA, in particular, on real functions defined by them. We state all the results for level automata only, although some of them are valid in general as well.

We start with closure properties.

LEMMA 3. *The family of real functions defined by level automata is closed under the operation of sum, product, and multiplication by a constant.*

Proof. For the first two operations the ordinary constructions of automata to recognize the union and the intersection work here as well. The closure under the last operation is due to the fact that we can multiply the initial distribution by a constant (so that here also a negative initial distribution must be allowed if the constant is negative). □

Our next representation is a straightforward consequence of the definitions.

LEMMA 4. *For each level automaton A there exists a finite number of line automata $A_1, \ldots, A_n$ such that $f_A = \sum_{i=1}^n f_{A_1}$.*

Our next lemma provides a normal form for level automata that turns out to be very important in § 6.

LEMMA 5. *For each level automaton A there exists an average preserving level automaton A_{ap} such that f_A and $f_{A_{\mathrm{ap}}}$ coincide.*

Proof. We prove the lemma only in case $\Sigma = \{0, 1\}$; the general proof is similar. Since each level automaton can be decomposed into a finite number of line automata, we may assume that A is a line automaton. Let its degree be t. Consequently, for $i = 0, \ldots, t$, there exists exactly one state of degree i, say q_i. Let us fix i and set

$$k = \frac{2 - W(q_i, 0, q_i) - W(q_i, 1, q_i)}{\sum_{p \neq q_i} W(q_i, 0, p) + \sum_{p \neq q_i} (q_i, 1, p)}. \tag{7}$$

Now, we will modify A to obtain the automaton A' as follows:

(i) The weight of each nonloop transition starting from q_i is multiplied by k.

(ii) If $i < t$, then the weight of each nonloop transition ending at q_i is multiplied by k^{-1}, otherwise $I(q_i)$ is multiplied by k^{-1}.

By (7), k is positive and chosen in such a way that the condition for q_i to be average preserving is satisfied.

We have to analyze the effects of the above modification to the computations of A. Let w be an arbitrary ω-word and w_n its prefix of length n. Assuming as above that the states of A and A' are ordered from q_t to q_0, we define

$$a_n(j) = P_A(w_n) \cdot \eta_j \quad \text{for } j = 0, \ldots, t$$

and

$$a'_n(j) = P_{A'}(w_n) \cdot \eta_j \quad \text{for } j = 0, \ldots, t,$$

where η_j is the vector having 1 as the jth component and 0 elsewhere.

Then it follows from the facts that A and A' are level automata and from the modifications (i) and (ii) that

$$\begin{aligned} a_n(j) &= a'_n(j) \quad \text{for } j > i, \\ a_n(i) &= \frac{1}{k} a'_n(i), \\ a_n(j) &= a'_n(j) \quad \text{for } j < i. \end{aligned}$$

Consequently (for details cf. (i)–(iii) in the proof of Lemma 6) we have

$$f_A(w) = f_{A'}(w)\,. \tag{8}$$

Note, however, that $F_A(w_n) \neq F_A(w_n)$ unless $k = 1$.

Now, the result follows easily. Since the state of degree 0 satisfies, by the definition of a level automaton, the average preserving condition, we first do the above modification to the state of degree 1. Then we repeat the process in the new automaton for the state of degree 2 and so on. Observe that in the last stage when doing (ii) we change the initial distribution from $(1, 0, \ldots, 0)$ to $(k^{-1}, 0, \ldots, 0)$. The automaton A_{ap} thus obtained is clearly average preserving, and moreover by (8), $f_{A_p} = f_A$. □

Lemma 5 deserves a few comments. On one hand, we have a normal form for functions on Σ^ω, or equivalently on $[0, 1]^d$, defined by level automata. On the other hand, Lemma 2 shows that average preserving automata define only the average preserving functions on $\Sigma^\star$, hence clearly not all the functions on $\Sigma^\star$ defined by level automata.

The proof of Lemma 5 was based on the idea of multiplying certain weights by suitable numbers in order to make a particular state to fulfill the average preserving condition. Another technique would be to increase the weights of some transitions by adding a constant to these weights. This addition can be compensated by changing the initial distribution at the state considered. In fact, it may get a negative value. We leave the details to the reader. We only note that by this method we can for any level automaton A find an average preserving level automaton A_{ap} defining the same real function as A. Similarly, we can show that any level automaton can be replaced by a 0- or 1-faithful automaton defining the same real function.

The next three lemmas deal with the continuity of the functions $\hat{f}_A$, where A is a given level automaton.

LEMMA 6. *The value $f_A(w)$ is defined* (*and finite*) *for each $w \in \{0, 1\}^\omega$. Consequently, $\hat{f}_A$ is continuous at any point x having only an infinite binary representation.*

Proof. Let A be a level automaton of degree t. Since A can be decomposed into a finite number of line automata of degree at most t, we can actually assume that A is a line automaton. Further, we can assume that its initial distribution is $(1, 0, 0, \ldots, 0)$ and that the ith component of this vector corresponds to the state of degree $t + 1 - i$ for $i = 1, \ldots, t + 1$. Let η_i be the $(t + 1)$-dimensional column vector such that its ith component equals to 1 and others to 0.

We define a new line automaton $A_{\max}$ of degree t as follows. Let α equal the maximum of the weights of the loops of the states having positive degree in A and β equal the total maximum of weights of A. Then in $A_{\max}$ all the loops at the states of a positive degree have the weight α, the loops at the state of degree 0 have the weights 1, and all the other weights equal β, in particular different from 0. Clearly, $A_{\max}$ is a well-defined line automaton and

$$0 \le P_A(u) \cdot \eta_i \le P_{A_{\max}}(u) \cdot \eta_i, \quad \text{for } i = 1, \ldots, t+1 \text{ and } u \in \Sigma^\star. \tag{9}$$

We prove that for any given $w \in \{0, 1\}^\omega$

$$\lim_{n\to\infty} P_A(\mathrm{pref}_n(w)) \cdot \eta_i = 0 \quad \text{for } i = 1, \ldots, t \tag{10}$$

and

$$\lim_{n\to\infty} P_A(\mathrm{pref}_n(w)) \cdot \eta_{t+1} \text{ exists.} \tag{11}$$

This clearly implies the first sentence of the lemma (and shows also that we can instead of the final distribution of the form $(1, \ldots, 1)$ use the distribution η_{t+1}).

To conclude (10) it is, by (9), enough to prove this for the automaton $A_{\max}$. This, in turn, is seen as follows. Define, for $i = 1, \ldots, t + 1$

$$a_n(i) = P_{A_{\max}}(\mathrm{pref}_n(w)) \cdot \eta_i \quad \text{for } n \ge 0.$$

Clearly, for all $n \ge 0$, we have

$$\begin{aligned} a_{n+1}(0) &= \alpha a_n(0), \\ a_{n+1}(j) &= \alpha a_n(j) + \beta(a_n(j-1) + \cdots + a_n(0)), \quad \text{for } j \le t, \\ a_{n+1}(t+1) &= a_n(t+1) + \beta(a_n(t) + \cdots + a_n(0)). \end{aligned}$$

It follows that

(i) $\lim_{n\to\infty} a_n(0) = 0$,
(ii) $\lim_{n\to\infty} a_n(j) = 0$ for $j \le t$, and
(iii) the sequence $(a_n(t+1))_{n\ge 0}$ is bounded.

Now, (10) follows from (i), (ii), and (9). To conclude (11) we first note that the sequence $(P_A(\mathrm{pref}_n(w)) \cdot \eta_{t+1})_{n\ge 0}$ is monotonous, and hence the limit exists by (9) and (iii).

The second sentence is a straightforward consequence of the first and its above proof. In particular, the fact implied by (9) and (iii) that f_A is a bounded function is needed. We leave the details to the reader. □

COROLLARY 1. *Let A be a level automaton. Then $\hat{f}_A$ is continuous if and only if it is continuous at the points that have a finite binary representation.*

Our next example shows that $\hat{f}_A$ can be noncontinuous at all points having a finite binary representation; cf. also Example 4 in §4.

Example 1. Consider the automaton A shown in Fig. 1. Observe that it slightly violates condition (ii) in the definition of the level automaton.

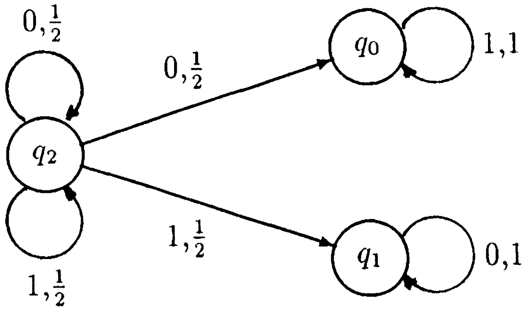


FIG. 1. *Automaton A of Example* 1.

Assume that the state q_2 is initial and q_0 and q_1 are final. Then

$$f_A(v01^\omega) = 2^{-|v|-1} = f_A(v10^\omega) \quad \text{for each } v \text{ in } \Sigma^\star$$

and

$$f_A(w) = 0 \quad \text{for each } w \text{ in } \Sigma^\omega.$$

Consequently, $\hat{f}_A$ is continuous exactly at those points that have only infinite binary representations. The graph of $\hat{f}_A$ is shown in Fig. 2.

Our next lemma shows that the obvious necessary condition for $\hat{f}_A$ to be continuous at the point 1/2 is also sufficient.

LEMMA 7. *The function $\hat{f}_A$ is continuous at the point* 1/2 *if and only if*

$$f_A(01^\omega) = f_A(10^\omega).$$

Proof. The necessity is obvious. The sufficiency follows from the proof of Lemma 6, in particular, that of the second sentence. □

LEMMA 8. *Let $x \in (0,1)$ have a finite binary representation, say, x is represented by the ω-words $v10^\omega$ and $v01^\omega$. Then $\hat{f}_A$ is continuous at x if and only if $\hat{f}_{A_v}$ is continuous at* 1/2 *where A_v is the WFA A with the initial distribution $P_A(v)$.*

Proof. Obvious by Lemma 7 and the identity

$$P_A(vu) = P_{A_v}(u) \quad \text{for all } u \text{ in } \Sigma^\star. \qquad □$$

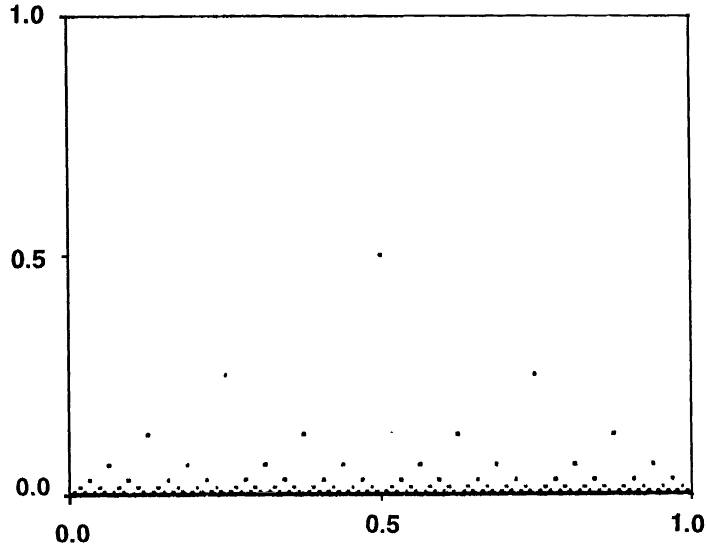


FIG. 2. *The graph of* $\hat{f}_A$ *of Example* 1.

By Lemma 8 we reduced the question of the continuity of a given $\hat{f}_A$ at a point with a finite binary representation to the same question at the point 1/2 of another function that is defined by the *same* automaton but different initial distribution. Typically, functions defined by level automata are continuous at 1/2 if and only if they are continuous at points having finite binary representations. However, this is not true in general as shown by the following example.

Example 2. Consider any level automaton A with initial state q_t and final state q_0 defining a noncontinuous function. Extend A to A' by adding to A a new state q_{t+1} and transitions

$$q_{t+1} \xrightarrow{0,1} q_t \quad \text{and} \quad q_{t+1} \xrightarrow{1,\hat{f}_A(1)} q_0.$$

Then clearly A' with the initial and final states q_{t+1} and q_0, respectively, defines the function that is continuous at 1/2 but not everywhere.

Our next example shows that even the very same level automata can define continuous and discontinuous functions depending on the initial distribution.

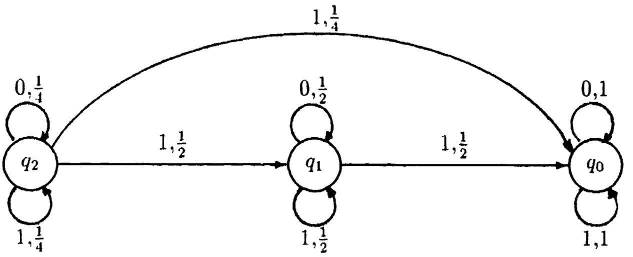
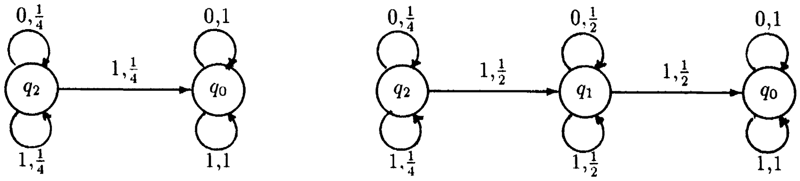


FIG. 3. *Line automaton A of Example* 3.

Example 3. Consider the line automaton A with the initial distribution $(1, 0, 0)$ shown in Fig. 3. As we shall see in §5, $\hat{f}_A(x) = x^2$ and thus $\hat{f}_A$ is continuous. However, A decomposes into two level automata, A_1 and A_2, shown in Fig. 4 in such a way that they



Automaton A_1

Automaton A_2

FIG. 4.

define noncontinuous functions at points having finite representations; cf. §4. Consequently, the level automaton A_3 shown in Fig. 5 is an example of such an automaton that with the initial distribution $(1, 0, 0, 0, 0)$ defines a continuous function and with the initial distribution $(0, 1, 0, 0, 0,)$ defines a noncontinuous function.

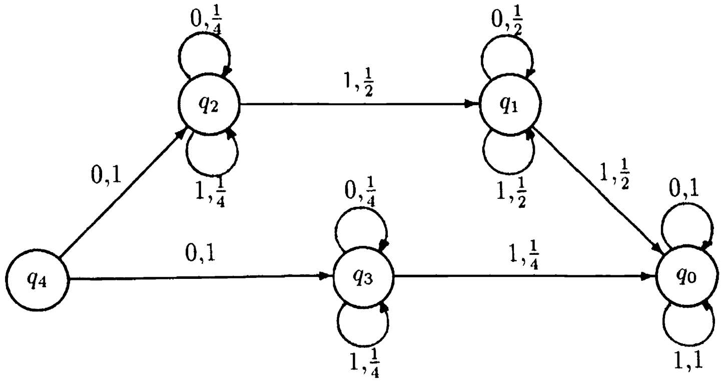


FIG. 5. *Automaton A_3.*

The above lemmas and examples show that although we can characterize the continuity of $\hat{f}_A$ at the point having a finite binary representation (or, in fact, at any point) in terms of the continuity of the very related function at the point $1/2$, it is not possible to characterize the continuity of $\hat{f}_A$ in terms of its continuity at the point $1/2$. However, our results allow us to prove the following important result.

THEOREM 1. *For a given level automaton A with rational weights it can be decided whether or not $\hat{f}_A$ is continuous.*

Proof. Assume that A has t states. Consider the set

$$X = \{P_A(v) \mid v \in \Sigma^\star\} \subseteq \mathbb{R}_+^t.$$

Let B be a base of the vector space over $\mathbb{R}$ generated by X. Clearly, its dimension is at most t, and such a base can be found effectively as follows. Consider a lexicographic order of $\Sigma^\star : u_0 = \varepsilon, u_1, u_2, \ldots$. Check systematically whether $P_A(u_i)$ is linearly independent of the vectors $P_A(u_j)$, $j < i$, as long as for some $k \geq 0$ no $u_i, i > k$, is linearly independent. Let the maximal set of linearly independent words thus obtained be $v_1, \ldots, v_s$. Then we can choose $B = \{P_A(v_i) | i = 1, \ldots, s\}$. For $i = 1, \ldots, s$ let A_i be the automaton A with the initial distribution $P_A(v_i)$.

CLAIM. *$\hat{f}_A$ is continuous if and only if each of the functions $\hat{f}_{A_i}$ for $i = 1, \ldots, s$ is continuous at the point $1/2$.*

Proof of the claim. If $\hat{f}_A$ is continuous, then in particular it is continuous at points represented by the words $v_i 10^\omega$ for $i = 1, \dots, s$. Now, the continuity of $\hat{f}_{A_i}$ at the point 1/2 follows since $f_A(v_i w) = f_{A_i}(w)$ for all $i = 1, \dots, s$ and $w \in \{0, 1\}^\omega$.

Conversely, assume that each of the functions $\hat{f}_{A_i}$ is continuous at the point 1/2. By Lemma 6, it is enough to show that $\hat{f}_A$ is continuous at an arbitrary point x having a finite binary representation, say, x is represented by $v10^\omega$. Now, by the choice of B there exist numbers α_i, for $i = 1, \dots, s$, such that

$$P_A(v) = \sum_{i=1}^{s} \alpha_i P_A(v_i),$$

implying that

$$f_A(vw) = \sum_{i=1}^{s} \alpha_i f_{A_i}(w).$$

This, however, by our assumption and Lemmas 7 and 8, proves that f_A is continuous at the point x. This completes the proof of the claim.

In order to finish the proof of Theorem 1, we have to show how to decide, for a given level automaton A, whether $\hat{f}_A$ is continuous at the point 1/2 or, equivalently, to decide whether the following identity holds true:

$$f_A(10^\omega) = f_A(01^\omega).$$

This can be equivalently stated as the question whether

$$\lim_{n\to\infty} (F_A(10^n) - F_A(01^n) \doteq 0$$

or whether

$$\lim_{n\to\infty} \pi \begin{pmatrix} W_0 & \Theta \\ \Theta & -W_1 \end{pmatrix}^n \eta = 0, \tag{12}$$

where W_0 and W_1 are the matrices of A associated to letter 0 and 1, Θ is the zero matrix of the appropriate size, π is the vector whose first half is equal to the distribution $\pi_1 W_1$ and the second half to $\pi_1 W_0$, and η is the vector whose both halves are equal to the final distribution of A.

The validity of (12) is easy to test. Indeed, since the matrices involved are upper triangular, their eigenvalues are directly visible and hence a formula for the number $F_A(01^n) - F_A(10^n)$ can be computed. It is of the form

$$\Sigma_i P_i(n)\rho_i^n,$$

where P_i's are polynomials and ρ_i's are the weights of the loops of A. This completes the proof of Theorem 1. □

Observe that the claim in the proof of Theorem 1 provides a characterization of the continuity of $\hat{f}_A$.

As the second important result we prove the following.

THEOREM 2. *It can be decided whether two given level automata with rational weights define the same real function.*

Proof. For 0-faithful level automata this follows straightforwardly from the Equality Theorem of Eilenberg [10]. Indeed, for such two automata A_1 and A_2 we have

$$(13) \qquad F_{A_i}(u) = f_{A_i}(u0^\omega) \quad \text{for } i = 1, 2,$$

and hence the problem to decide the equivalence of the functions $\hat{f}_{A_i}$ and $\hat{f}_{A_2}$ on numbers having a finite binary representation reduces, by the Equality Theorem, to the question of testing whether

$$(14) \qquad F_{A_1}(u) = F_{A_2}(u),$$

for words u of length at most the sum of the numbers of the states of A_1 and A_2. Of course, this can be done. Finally, the same test reveals, by Lemma 6, whether the identity

$$f_{A_1}(w) = f_{A_2}(w)$$

holds true for all ω-words w as well.

If A_1 and A_2 are not 0-faithful, then the above proof has to be modified as follows. Now, we first decide whether

$$(15) \qquad f_{A_1}(u0^\omega) = f_{A_2}(u0^\omega) \quad \text{for all } u \in \{0, 1\}^\star.$$

This is reduced, as in the Equality Theorem, using the very same idea as in the proof of Theorem 1, to test (15) on a finite number of words u only. Hence, the considerations at the end of Theorem 1 guarantee that we can decide whether (15) holds true, that is whether

$$(16) \qquad \hat{f}_{A_1}(x) = \hat{f}_{A_2}(x) \quad \text{for all } x \text{ having a finite representation.}$$

But then as above, by Lemma 6, we can also test whether (16) holds true for all x in $[0, 1]$. □

4. Line automata of degree one. In this section we investigate functions defined by WFA of the form shown in Fig. 6, where all the weights are positive and the initial and final distributions are $(1, 0)$ and $(1, 1)$, respectively. We want A to define a bounded function $\hat{f}_A : [0, 1] \to \mathbb{R}$. Consequently, the weights α, β, ζ and δ are at most 1. On the other hand, if all of them are smaller than 1, then A defines the zero-function. Therefore, we assume that $0 \leq \alpha, \beta, \zeta, \delta \leq 1$ and some of them equal to 1. The most natural case is that $\varepsilon = \delta = 1$, which corresponds to the case of line automata of degree 1. We shall analyze this case in detail.

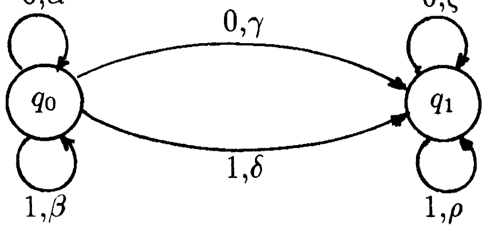


FIG. 6. *A general* WFA *A of* §4.

We propose two questions:

(I) Under which conditions is $\hat{f}_A$ continuous?

(II) Under which conditions is $\hat{f}_A$ *smooth*, that is, have all derivatives?

In order to answer (I) we start to consider the continuity at point 1/2. Since 1/2 has the representations 01^ω and 10^ω, we must have

$$f_A(01^\omega) = f_A(10^\omega). \tag{17}$$

Now, $P_A(01^n) = (\alpha\beta^n, \alpha\delta \sum_{i=0}^{n-1} \beta^i + \gamma)$ so that if $\beta < 1$, we obtain

$$\begin{aligned} f_A(01^\omega) &= \lim_{n\to\infty} \left(\alpha\beta^n + \alpha\delta \sum_{i=0}^{n-1} \beta^i + \gamma\right) \\ &= \alpha\delta\frac{1}{1-\beta} + \gamma. \end{aligned} \tag{18}$$

Similarly assuming that $\alpha < 1$ we have

$$f_A(10^\omega) = \beta\gamma\frac{1}{1-\alpha} + \delta.$$

Hence, (17) yields the condition

$$\alpha\delta\frac{1}{1-\beta} + \gamma = \beta\gamma\frac{1}{1-\alpha} + \delta \tag{19}$$

or, equivalently,

$$(\alpha+\beta-1)(\delta(1-\alpha)-\gamma(1-\beta)) = 0.$$

It follows that under the assumption $\alpha, \beta < 1$ the continuity of $\hat{f}_A$ at 1/2 requires that one of the following two conditions is satisfied:

(i) $\alpha + \beta = 1$,

(ii) $\delta(1-\alpha) = \gamma(1-\beta)$.

These considerations can be reversed by Lemma 7 or straightforward calculations based on the implication: If $P_A(v) = (A, B)$, then $P_A(v0) = (\alpha A, \gamma A + \zeta B)$ and $P_A(v1) = (\beta A, \delta A + \rho B)$. Consequently, conditions (i) and (ii) (under the assumption $\alpha, \beta < 1$) characterize the continuity of $\hat{f}_A$ at the point 1/2.

The continuity of $\hat{f}_A$ at the point x that has a finite binary representation $v1$ can be analyzed by considering instead of (17) the condition

$$f_A(v01^\omega) = f_A(v10^\omega). \tag{20}$$

Now, if $P_A(v) = (A, B)$, then (19) can be written as

$$A(\alpha\delta\,\frac{1}{1-\beta} + \gamma) + B = A(\beta\gamma\,\frac{1}{1-\beta} + \gamma) + B, \tag{21}$$

which is equivalent to (18), since clearly $A > 0$. Consequently, the continuity of $\hat{f}_A$ at x is characterized by exactly the same conditions as that at 1/2, namely, (i) and (ii).

The continuity of $\hat{f}_A$ at the point x that does not have a finite binary representation follows from Lemma 6.

All in all, we have established the following.

THEOREM 3. *The automaton A with $0 \leq \alpha, \beta < 1$, $0 \leq \gamma, \delta$, and $\zeta = \rho = 1$ defines a continuous function $\hat{f}_A$ if and only if either* (i) *or* (ii) *holds true.*

Observe that conditions (i) and (ii) are not exclusive and that the first condition does not give any restriction for γ and δ, while the second one fixes their ratio. We shall see in a moment that (ii) gives only constant functions. Condition (i), in turn, gives both linear functions and something that can be called fractal-type functions.

In order to complete the analysis of the continuity of $\hat{f}_A$ we have to consider the excluded cases $\alpha = 1$ or $\beta = 1$. If $\beta = 1$, then in order to keep $\hat{f}_A$ bounded, necessarily $\alpha = 0$ or $\delta = 0$. In the first case (19) can be rewritten as $\gamma = \gamma + \delta$ so that $\delta = 0$, and hence $\hat{f}_A(1) = 1$ and $\hat{f}_A(x) = \gamma$ for $x < 1$. In the second case (19) can be rewritten as $\gamma = \gamma(1-\alpha)^{-1}$, so that $\alpha = 0$, and we are back in the first case. Finally, if both $\beta = 1$ and $\alpha = 1$, then, as above, we have $\gamma = \delta = 0$, and hence $\hat{f}_A$ is the constant function $\hat{f}_A = 1$.

In the above we completely answered question (I) (in the case $\zeta = \rho = 1$). Now, we turn to consider question (II), and we can restrict our considerations to the case when $\alpha, \beta < 1$.

Let us assume that $\hat{f}_A$ is smooth, in particular, that it has a derivative at the point $1/2$. This means that the limits

$$\lim_{n\to\infty} \frac{f_A(10^\omega) - f_A(01^n0^\omega)}{2^{-(n+1)}}$$

and

$$\lim_{n\to\infty} \frac{f_A(10^n1^\omega) - f_A(01^\omega)}{2^{-(n+1)}}$$

exist and are equal.

We already computed that

$$f_A(01^\omega) = \alpha\delta\frac{1}{1-\beta} + \gamma.$$

Further, we obtain (cf. (18) and (21)).

$$f_A(01^n0^\omega) = \alpha\beta^{n-1}\left(\alpha\delta\frac{1}{1-\beta} + \gamma\right) + \alpha\delta\sum_{i=0}^{n-2}\beta^i + \gamma.$$

Therefore, recalling that we are working under the assumption that $\hat{f}_A$ is continuous, we obtain

$$(22) \qquad \lim_{n\to\infty} \frac{f_A(10^\omega) - f_A(01^n0^\omega)}{2^{-(n+1)}}$$

$$= \lim_{n\to\infty} 2^{n+1}\left(\alpha\delta\frac{\beta^{n-1}}{1-\beta} - \alpha^2\delta\frac{\beta^{n-1}}{1-\beta} - \alpha\gamma\beta^{n-1}\right)$$

$$= \begin{cases} 0 & \text{if (i) is satisfied and } \beta < 1/2, \\ 4\alpha((2-2\alpha)\delta - \gamma) & \text{if (i) is satisfied and } \beta = 1/2, \\ 0 & \text{if (ii) is satisfied,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Similarly,

$$(23)\qquad \lim_{n\to\infty} \frac{f_A(10^n1^\omega) - f_A(01^\omega)}{2^{-(n+1)}}$$

$$= \begin{cases} 0 & \text{if (i) is satisfied and} \, \alpha < 1/2 \, , \\ 4\beta(\delta - (2-2\beta)\gamma) & \text{if (i) is satisfied and} \, \alpha = 1/2 \, , \\ 0 & \text{if (ii) is satisfied ,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

As in the case of continuity, the above considerations can be extended from point 1/2 to any point having a finite binary representation. Then in formulas (4) and (23) the right-hand sides are replaced by constant multiples of those in (4) and (23); cf. considerations in obtaining (21). Whether $\hat{f}_A$ has derivatives at the points that do not have a finite binary representation is discussed later.

Now, we determine the existence of the derivative of $\hat{f}_A$ at the point 1/2 (or equivalently at any point with a finite binary representation). We have two main cases depending on which of the continuity conditions (i) and (ii) is satisfied.

Case I. Condition (i) holds true. There are three subcases.

(a) $\beta < 1/2$. Now, either (ii) is satisfied as well or $\alpha > 1/2$ and the fourth alternative of (23) takes place. The first possibility is treated in Case II. In the second possibility the derivative (in fact, also the right derivative) of $\hat{f}_A$ at 1/2 is undefined.

(b) $\beta = 1/2$, and hence also $\alpha = 1/2$. Now, the left and right derivatives of $\hat{f}_A$ at 1/2 equal $2(\delta - \gamma)$ and hence

$$(24)\qquad \hat{f}'_A(1/2) = 2(\delta - \gamma).$$

Observe that this is consistent with the possibility that (ii) also holds true. Namely, then $\gamma = \delta$ and so (8) yields $\hat{f}'_A(1/2) = 0$ as it should by the third lines of (4) and (23). Now, the automaton is of the form shown in Fig. 7.

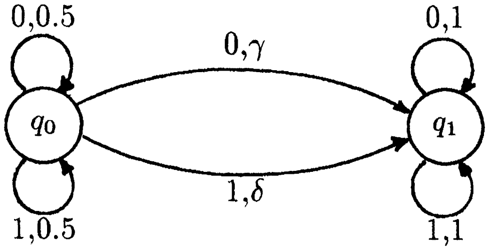


FIG. 7. *Automaton* A_1.

Since (24) holds true not only at 1/2 but also at any point having a finite binary representation, it strongly supports that $\hat{f}_A$ is a linear function. This can be confirmed easily from the form of the WFA, which at the same time proves that (24) holds true for all x in (0, 1). Indeed, we claim that

$$\hat{f}_A(x) = 2(\delta - \gamma)x + 2\gamma \quad \text{for}\, x \in [0, 1].$$

Let $w = a_1a_2\ldots$ be the standard binary representation of the number x in [0, 1], i.e.,

$$x = \sum_{i=1}^{\infty} a_i 2^{-i} \quad \text{with}\, a_i \in \{0, 1\}.$$

Then clearly,

$$\begin{aligned} f_A(w) &= \delta \sum_{a_i=1} 2^{-i+1} + \gamma \sum_{a_i=0} 2^{-i+1} \\ &= \delta \sum_{a_i=1} 2^{-i+1} - \gamma \sum_{a_i=1} 2^{-i+1} + \gamma \sum_{i=1}^{\infty} 2^{-i+1} \\ &= 2\delta x - 2\gamma x + 2\gamma, \end{aligned}$$

which proves the claim.

(c) $\beta > 1/2$ and (ii) is satisfied. This is symmetric to the second possibility of Case I(a).

Case II. Condition (ii) holds true. Now, the automaton, say A_2, is of the form shown in Fig. 8, where the case of $\alpha = 0$ is allowed. Using the method of Case I(b) one easily concludes that

$$\hat{f}_{A_2}(x) = 2\delta \quad \text{for all } x \text{ in } [0, 1].$$

Observe that this again is consistent with our computed value of the derivative of $\hat{f}_A$ at $1/2$. As a conclusion of the above we have proved:

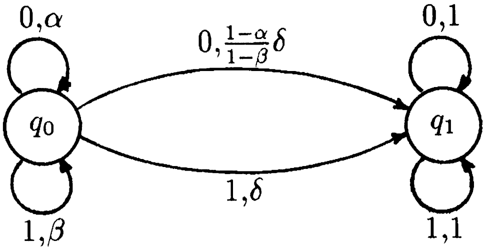


FIG. 8. *Automaton* A_2.

THEOREM 4. *The function* $\hat{f}_A$ *defined by the* WFA A *of Fig.* 6 *is smooth if and only if it is linear. Moreover, such a function is not a constant if and only if* $\alpha = \beta = 1/2$ *and* $\gamma \neq \delta$.

Actually, we proved a stronger result, as seen in the following theorem.

THEOREM 5. *The function* $\hat{f}_A$ *defined by the* WFA A *of Fig.* 6 *has the derivative at the point* $1/2$ *if and only if it is a linear on the interval* $(0, 1)$.

We have completely characterized both continuous and smooth functions defined by automaton A. Most typically, such functions are not continuous like the one given in Example 4, and they are smooth only in very exceptional cases as shown by Theorem 4. What is most interesting is that there exist such functions that are continuous but not smooth. As we saw such functions do not possess a derivative at any point having a finite binary representation. We analyze such functions in more details in Example 5.

Example 4. The choice $\alpha = 1/2$, $\beta = 1/4$, $\gamma = 1$, and $\delta = 0$ in the automaton of Fig. 6 gives an example of a noncontinuous function, in fact, of the function that is not continuous at any point having a finite binary representation. Its graph is illustrated in Fig. 9.

Example 5. Let us analyze the function defined by the automaton B shown in Fig. 10.

Let the initial and final distributions of B be the standard ones, $(1, 0)$ and $(1, 1)$. Then for any finite word u

$$P_B(u) = (a(u), b(u)), \tag{25}$$

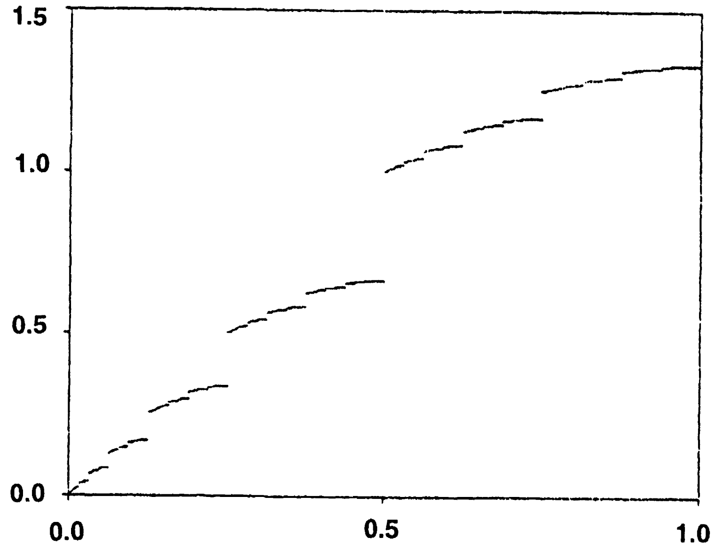


FIG. 9. *The graph of the* WFA *of Example* 4.

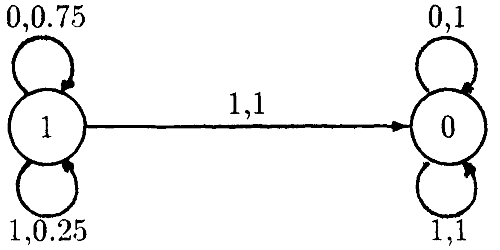


FIG. 10. *Automaton B of Example* 5.

where

$$a(u) = \frac{3^{|u|_0}}{4^{|u|}} \tag{26}$$

and $b(u)$ is defined recursively on prefixes v of u as follows:

$$\begin{aligned} b(\varepsilon) &= 0 \\ b(vt) &= \begin{cases} b(v) & \text{if } t = 0, \\ b(v) + a(v) & \text{if } t = 1. \end{cases} \end{aligned} \tag{27}$$

By our general considerations $\hat{f}_B$ is continuous; cf. Fig. 14 showing its graph. Moreover, the above formulas show that it is monotonous and for any $w \in \{0, 1\}^\omega$

$$\hat{f}_B(w) = \lim_{n \to \infty} b(w_n), \tag{28}$$

where w_n denotes the prefix of w of the length n.

We consider the derivability of $\hat{f}_B$ in two particular points:

$$\begin{aligned} w_1 &= (01)^\omega, \quad \text{and} \\ w_2 & \quad \text{satisfying for each prefix u of } w_2, |u|_0 \geq 2|u|_1. \end{aligned}$$

The existence of the derivative at $\hat{w}_1$ is concluded as follows. First we calculate that

$$\begin{aligned} f_B(w_1) &= \tfrac{3}{4} \textstyle\sum_{i=0}^{\infty} (\tfrac{1}{4} \cdot \tfrac{3}{4})^i = \tfrac{3}{4} \cdot \tfrac{16}{13} = \tfrac{12}{13}, \\ f_B((01)^{n-1}0^\omega) &= \tfrac{12}{13}(1 - (\tfrac{3}{16})^{n-1}) = \tfrac{12}{13} - \tfrac{12}{13}(\tfrac{3}{16})^{n-1}, \\ f_B((01)^{n-1}1^\omega) &= \tfrac{12}{13}(1 - (\tfrac{3}{16})^{n-1}) + \tfrac{4}{3} \cdot (\tfrac{3}{16})^{n-1} = \tfrac{12}{13} + \tfrac{16}{39}(\tfrac{3}{16})^{n-1}. \end{aligned}$$

Therefore, by the monotonicity of $\hat{f}_B$, for any $w = (01)^{n-1}w'$ we have

$$|f_B(w_1) - f_B(w)| \leq \tfrac{16}{39}(\tfrac{3}{16})^{n-1}.$$

But these w's are exactly those that satisfy $|\hat{w} - \hat{w}_1| \leq 4 \cdot 4^{-n}$ and hence $\hat{f}'_B(\hat{w}_1) = 0$.

The nonexistence of the derivative at the point $\hat{w}_2$ is seen as follows. If w_2 contains ultimately only 1's, our earlier general considerations apply. Otherwise we consider the word $w_2(n)$ that is obtained from w_2 by changing the nth symbol from 0 to 1 assuming that it is originally 0. Then

$$|\widehat{w_2(n)} - \hat{w}_2| = 2^{-n}.$$

Moreover, by formulas (25), (27), and (28), we conclude that

$$f_B(w_2(n)) - f_B(w_2) \geq a(\text{pref}_{n-1}(w_2)),$$

and hence, by (26) and the special form of w_2, we have

$$f_B(w_2(n) - f_B(w_2) \geq \frac{3^{2/3n}}{4^n}.$$

Consequently,

$$\frac{f_B(w_2(n)) - f_B(w_2)}{\widehat{w_2(n)} - \hat{w}_2} \geq 2^n(\frac{\sqrt[3]{2}}{4})^n > (1,03)^n,$$

which proves our claim.

As a conclusion we note that $\hat{f}_B$ is an example of a continuous function that is not at all smooth. Indeed, the set of points where it does not have a derivative is not only dense but also uncountable. Observe also that our choice for the decomposition $\alpha + \beta = 1$ is not essential.

5. Generation of polynomials. In the previous section we analyzed what it means for a function defined by a WFA to have certain regularity properties. As a byproduct we concluded that all linear functions (in the interval [0, 1]) are definable by very simple two-state WFA, which is by the line automata of degree one. Here we take a synthesizing approach and show that all polynomials in one or several variables can be defined as functions of WFA.

THEOREM 6. *For each polynomial $P(x)$ with nonnegative coefficients and of degree n there exists a line automaton A_n of degree n such that $f_{A_n}(x) = P(x)$ for all x in* [0, 1].

The restriction of coefficients can be dropped if negative values in the initial distribution are allowed.

Proof. We first prove the result for powers of x, that is, for polynomials of the form x^n. For these we prove an even stronger result:

CLAIM. The polynomial x^n, for $n \geq 0$, can be defined by a line automaton A_n of degree n of the following form: The initial and final distributions of A_n are $(1, 0, \ldots, 0)$ and $(1, 1, \ldots, 1)$, and denoting by $\{q_n, \ldots, q_0\}$ the state set of A_n the transitions are

(i) $q_i \xrightarrow{j,2^{-i}} q_i$, for $j = 0, 1$ and $i = 0, \ldots, n$,

(ii) $q_i \xrightarrow{1,q^{-i}\binom{i}{t}} q_{i-t}$ for $i = 1, \ldots, n$ and $t = 1, \ldots, i$.

Proof of the claim. For $n = 0$ the claim is trivial, and for $n = 1$ it was established in the previous section. Indeed, function x is defined by the automaton shown in Fig. 11 with standard initial and final distributions. Assume now inductively that the claim has been proved for a fixed n. For $n + 1$ the automaton A_{n+1} is of the form shown in Fig. 12.

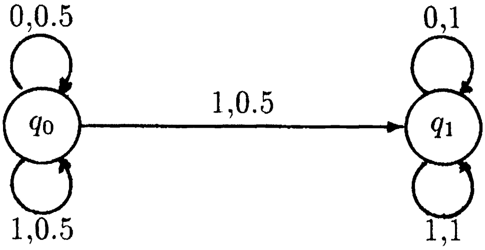


FIG. 11. *Line automaton A_1.*

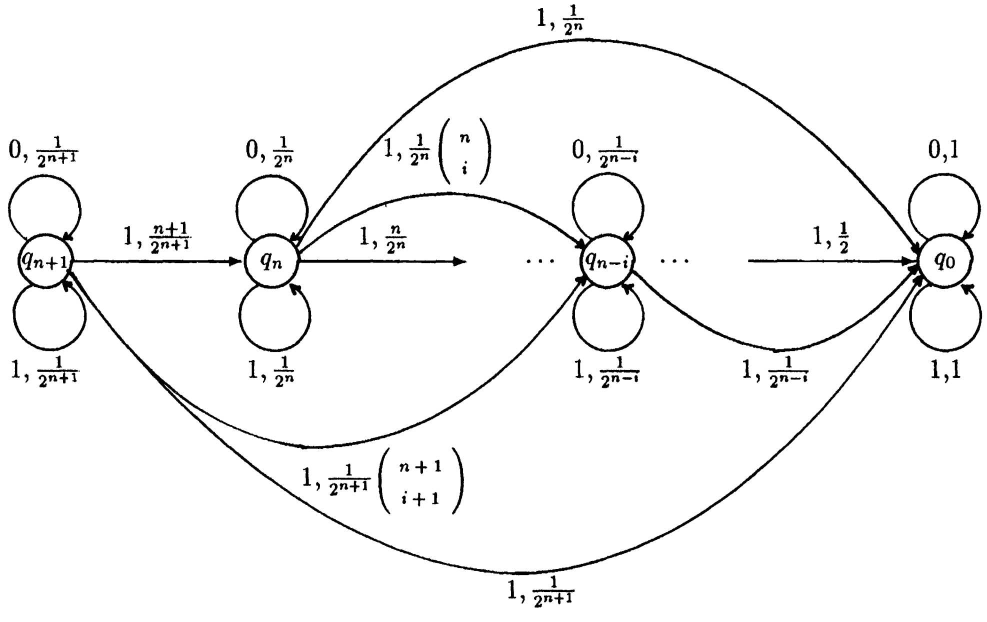


FIG. 12. *Line automaton A_n.*

By construction the automaton A_{n+1} has the following nice properties.

(i) For each $i = 0, \dots, n+1$, the sum of weights of transitions starting from q_i and labeled by 1 equals 1 by the binomial formula. Therefore, f_{A_i} is 1-faithful.

(ii) A_i for $i \leq n$ is obtained from A_{n+1} by deleting states $q_{n+1}, \dots, q_{i+1}$ and transitions connected to these.

Let x be an arbitrary number in [0, 1] and w its standard binary representation so that $x = \hat{w}$. By construction of A_{n+1}

$$f_{A_{n+1}}(0w) = \frac{1}{2^{n+1}} f_{A_{n+1}}(w)$$

and

$$f_{A_{n+1}}(1w) = \frac{1}{2^{n+1}} f_{A_{n+1}}(w) + \sum_{i=0}^{n} \frac{1}{2^{n+1}} \binom{n+1}{i+1} f_{A_{n-1}}(w).$$

Since $\widehat{0w} = \frac{1}{2}x$ and $\widehat{1w} = \frac{1}{2}x + \frac{1}{2}$, these equations can be rewritten, by induction assumption, as

$$\hat{f}_{A_{n+1}}\left(\frac{1}{2}x\right) = \frac{1}{2^{n+1}} \hat{f}_{A_{n+1}}(x) \tag{29}$$

and

$$\hat{f}_{A_{n+1}}\left(\frac{1}{2}x + \frac{1}{2}\right) = \frac{1}{2^{n+1}} \hat{f}_{A_{n+1}}(x) + \sum_{i=0}^{n} \frac{1}{2^{n+1}} \binom{n+1}{i+1} x^n. \tag{30}$$

Moreover, by (i),

$$\hat{f}_{A_{n+1}}(1) = 1. \tag{31}$$

Formulas (29) and (30) hold true for all x in [0, 1]. So together with (31) they imply that

$$\hat{f}_{A_{n+1}}(z) = z^{n+1}$$

for all z having a finite binary representation. Consequently, since the set of such points is dense in [0, 1], the claim follows if we show that $\hat{f}_{A_{n+1}}$ is continuous. This is seen as follows. By induction hypothesis and the claim in the proof of Theorem 1, $\hat{f}_{A_{n+1}}$ is continuous at points having finite binary representations if and only if it is continuous at the point 1/2. Therefore, the continuity of $\hat{f}_A$ follows from Lemmas 6 and 7 together with the equations

$$f_{A_{n+1}}(10^\omega) = \frac{1}{2^{n+1}} = f_A(01^\omega),$$

the first of which is obvious and the second one is due to (i). This completes the proof of the claim. □

After establishing the claim, Theorem 6 follows easily. Polynomial $a_{n+1}x^{n+1} + \cdots + a_0$, with $a_i \in \mathbb{R}_+$, is obtained by using the automaton A_{n+1} with the initial distribution $(a_{n+1}, \dots, a_0)$. Obviously, the second sentence of the theorem follows as well.

In the above proof we constructed directly a WFA for a given polynomial. In this construction there are several interesting phenomena. First, for any polynomial of degree n, or in fact of degree at most n, we can use the *same* line automaton A_n, only with different initial distributions. A related observation is that our WFA for x^n can be extended by introducing a new state and transitions for it to a WFA for the function x^{n+1}.

Second, the structure of A_{n+1} is very interesting. For each state there are equally weighted loops for both letters, the weights being inverses of powers of 2 : $2^{-(n+1)}, 2^{-n}, \dots, 2^{-1}, 1$. All states connected by transitions are connected by transitions labeled by 1, and all states are acyclically connected in this way. Moreover, for each state the sum of weights starting from this state and labeled by 1 equal 1, as already noted in (i). Consequently, our WFA is 1-faithful. Of course, it would be possible to change the roles of the input letters and make it 0-faithful. Now, with the initial distribution $(1, 0, \dots, 0)$, we obtain functions that at 0 assume value 0 and at 1 value 1; by the change we would get functions that at 0 get value 1 and at 1 value 0.

There is another way to find a WFA defining a given polynomial function P of degree n. This is based on the closure properties of WFA. We can start from a WFA defining the linear function x, such as A_1. Then for x^n we get a WFA as an n-fold product of A_1, and finally as a sum we obtain a WFA automaton for P. Now, the resulting automaton is not a line automaton but a level automaton of degree n (possibly with negative values in the initial distribution). However, it also has several important properties of A_n. Indeed, for each state there are equally weighted loops for both letters, and the weights are as in A_n powers of inverses of two. Moreover, all the other transitions are labeled only by 1's with possibly different weights. Finally, the automaton is 1-faithful, if the starting automaton is so, and no states except the sink states are merged when summing the automata together.

Now, we turn to consider the generation of polynomials over several variables by WFA. In addition to Theorem 6 we need one operation on automata that is quite natural but not widely used, namely, their *Cartesian product* [11]. We formulate it here for WFA. For $i = 1, 2$ let

$$A_i = (Q_i, \Sigma_i, W_i, I_i, T_i)$$

be a WFA. We define Cartesian product of A_1 and A_2 as a WFA

$$A = A_1 \times A_2 = (Q_1 \times Q_2, \Sigma_1 \times \Sigma_2, W, I, T)$$

where

$$I : Q_1 \times Q_2 \to \mathbb{R}, \qquad I(q_1, q_2) = (I_1(q_1), I_2(q_2));$$

$$T : Q_1 \times Q_2 \to \mathbb{R}, \qquad T(q_1, q_2) = (T_1(q_1), T_2(q_2));$$

and

$$W(p_1, p_2, a_1, a_2, q_1, q_2) = W_1(p_1, a_1, q_1) W_2(p_2, a_2, q_2).$$

Consequently, A defines a function $F_A : (\Sigma_1 \times \Sigma_2)^\star \to \mathbb{R}$, which is a rational function over the alphabet of the Cartesian product of the original alphabets. For example, if $\Sigma_1 = \Sigma_2 = \{0, 1\}$, this is exactly what we need in order to compute two-dimensional functions $\mathbb{R}^2 \to \mathbb{R}$, since a vector (u, v), with $u, v \in \{0, 1\}^\star$ and $|u| = |v|$, addresses a point in the square $[0, 1] \times [0, 1]$. Such a vector can be viewed, as we do, in a natural way as a word over the four-letter alphabet. Of course, if we want to compute functions $\mathbb{R}^d \to \mathbb{R}$ we take the n-fold Cartesian product of two WFA over $\{0, 1\}$, so that the alphabet Σ^d of the WFA has 2^d letters.

THEOREM 7. *For each polynomial $P(x_1, \dots, x_d)$ with nonnegative coefficients and defined in $[0, 1]^d$ there exists a WFA A defining it. Moreover, A can be chosen to have the level structure of depth equal to the maximum of the powers of variables in $P(x_1, \dots, x_d)$. The restriction on coefficients can be dropped out if negative values in the initial distributions are allowed.*

Proof. We consider first the case when P is of the form $P(x, y) = x^n y^m$. By Theorem 6, we can find line automata A_n and A_m such that

$$\hat{f}_{A_n}(x) = x^n \quad \text{and} \quad \hat{f}_{A_m}(y) = y^m .$$

We claim that the Cartesian product $A = A_n \times A_m$ computes P, that is,

$$\hat{f}_A(x, y) = x^n y^m , \tag{32}$$

or, equivalently, denoting by $\pi_i(w)$ the ith projection of $w \in (\{0, 1\} \times \{0, 1\})^\omega$, that is,

$$f_A(w) = f_{A_1}(\pi_1(w)) \cdot f_{A_2}(\pi_2(w))$$

with $\pi_1(w)$ and $\pi_2(w)$ representing x and y, respectively.

The proof of (32) is straightforward. Let $\gamma \in (\{0, 1\} \times \{0, 1\})^*$, and denote $u = \pi_1(\gamma)$ and $v = \pi_2(\gamma)$. As it is easy to see by induction, it follows from the construction of A that the weight caused by w to the state (p, q) of A equals

$$w_p(u)\, w_q(v) ,$$

where $w_p(u)$ and $w_q(v)$ define the weights of the states p of A_n and q of A_m caused by the words u and v, respectively. Consequently, for $w \in (\{0, 1\} \times \{0, 1\})^\omega$ we can write

$$f_A(w) = \lim_{n\to\infty} \sum_{p\in Q_1, q\in Q_2} w_p(u_n) \cdot w_q(v_n), \tag{33}$$

where u_n and v_n are the prefixes of length n of $\pi_1(w)$ and $\pi_2(w)$, respectively. Since Q_1 and Q_2 are finite, we conclude that

$$\begin{aligned} f_A(w) &= \sum_{p\in Q_1} \lim_{n\to\infty} w_p(u_n) \left(\sum_{q\in Q_2} \lim_{n\to\infty} w_q(v_n) \right) \\ &= \sum_{p\in Q_1} \lim_{n\to\infty} w_p(u_n) \cdot f_{A_2}(\pi_2(w)) \\ &= f_{A_1}(\pi_1(w)) \cdot f_{A_2}(\pi_2(w)) , \end{aligned}$$

as claimed.

Now, the proof of the general case is easy. Each monomial of P is definable by a certain product of line automata of depths equal to the power of the corresponding variable in the monomial. Such a product is, as is obvious, a level automaton of degree of the highest power of the variables. Hence, for any monomial of P we obtain a level automaton of an appropriate depth and having a suitable initial distribution (determined by the coefficient of the monomial). Moreover, it is a consequence of Theorem 6 that this level automata can be chosen the *same* for each monomial. Hence, the polynomial is definable, with a suitable initial (not necessarily positive) distribution, by a level automaton of degree the maximum of the powers of the variables in $P(x_1, \ldots, x_d)$. □

The above proof deserves two comments. Part of the proof of Theorem 7 was to show a closure property of WFA automata. It was formulated for particular level automata (to guarantee the existence of the limits) but actually holds in a much more general form. The other point worth noticing is that, as in Theorem 6, also here we get all polynomials up to certain degree from the same WFA having only different initial distributions. Moreover, this WFA possesses the level (but not necessarily line) structure of the depth of the maximal degree of variables in P. This property might have practical importance as well.

6. Integration. This section is devoted to prove the following result, where the integral means ordinary Riemann integral.

THEOREM 8. *For each level automaton A of degree n there exists a level automaton A_I of degree $n+1$ such that*

$$\hat{f}_{A_I}(x) = \int_0^x f_A(t)\,dt. \tag{34}$$

Moreover, given the automaton A the automaton A_I can be constructed by uniform rules.

Proof. Assume that A defining the function $\hat{f}_A : [0,1] \to \mathbb{R}$ is given. We first apply Lemma 5 in order to find an average preserving level automaton A_{ap} (of the same degree) defining $\hat{f}_A$. Next, we define a level automaton A_I of degree $n+1$ as follows:

(i) Its states are those of A_{ap} together with one new state called t.

(ii) Its transitions contain transitions of A_{ap} with weights divided by 2.

(iii) From each original state q of A_{ap} there is a transition in A_I to the state t labeled by 1 and with the weight $\frac{1}{2}\sum_{p\in Q} W(q,0,p)$, where W is the weight function of A_{ap} and Q is its set of states.

(iv) The weight 1.

It remains to prove that (34) holds true. We do that in two stages.

CLAIM 1. $f_{A_I}(w) = \sum_{i=1}^{\infty} \mathrm{Bin}(i)\cdot 2^{-i} F_{A_{\mathrm{ap}}}(\mathrm{pref}_{i-1}(w)\cdot 0)$, where $w = a_1a_2\ldots$ with $a_i \in \{0,1\}$ and $\mathrm{Bin}(i)$ in the ith bit of w.

Proof of Claim 1. Let η_{n+1} denote the $(n+1)$-dimensional vector such that its last component equals 1 and the others equal 0. Then by the construction of A_I we have

$$P_{A_I}(\varepsilon)\cdot\eta_{n+1} = 0$$

and for all $u \in \{0,1\}^*$ we have

$$P_{A_I}(u0)\cdot\eta_{n+1} = P_{A_I}(u)\cdot\eta_{n+1},$$

$$P_{A_I}(u1)\cdot\eta_{n+1} = P_{A_I}(u)\cdot\eta_{n+1} + 2^{-|u|-1}F_{A_{\mathrm{ap}}}(u0).$$

Since A_I is a level automaton, the value of f_{A_I} is obtained in the last state t only (although the values of $F_{A_{\mathrm{ap}}}$ are obtained by summing up the values of all states). Therefore Claim 1 follows from the above equalities.

CLAIM 2. $\int_0^x f_{A_{\mathrm{ap}}}(t)\,dt = \sum_{i=1}^{\infty} \mathrm{Bin}(i)\, 2^{-i} F_{A_{\mathrm{ap}}}(\mathrm{pref}_{i-1}(w)\cdot 0)$, where $w = a_1a_2\ldots$ is the standard representation of $x \in [0,1]$ and $\mathrm{Bin}(i)$ is the ith bit of w.

Proof of Claim 2. We write w in the form

$$w = u10^k1w' = v1w' \quad \text{for some } k \geq 0, \tag{35}$$

and denote by x_1 and x_2 the numbers having the representations $u1$ (or 0 if the shown k 0's are at the beginning of w) and $u10^k1$, respectively. We show that

$$I = \int_{x_1}^{x_2} f_{A_{\mathrm{ap}}}(t)dt = \frac{1}{2^{|v0|}}F_{A_{\mathrm{ap}}}(v0). \tag{36}$$

By the definition of the integration

$$I = \lim_{n\to\infty}\sum_{i=0}^{2^n-1}\frac{f_{A_{\mathrm{ap}}}(v0w_i0^{\omega})}{2^{|v0w_i|}}, \tag{37}$$

where w_i is the ith word in the lexicographic order of the words of length n. Now, by arguments of the proof of Lemma 6 we conclude the existence of the function $\varepsilon : \mathbb{N} \to \mathbb{R}_+$ such that

$$(38)\qquad | f_{A_{\mathrm{ap}}}(z0^{\omega}) - F_{A_{\mathrm{ap}}}(z) | \le \varepsilon(|z|), \quad \forall z \in \{0,1\}^*$$

and

$$(39)\qquad \lim_{n\to\infty} \varepsilon(n) = 0.$$

Finally, since $F_{A_{\mathrm{ap}}}$ is average preserving, we have

$$(40)\qquad \sum_{i=0}^{2^n-1} \frac{F_{A_{\mathrm{ap}}}(v0w_i)}{2^{|v0w_i|}} = \frac{F_{A_{\mathrm{ap}}}(v0)}{2^{|v0|}}.$$

Now, (36) follows straightforwardly from (37)–(40). □

Observe that the rules (i)–(iv) to construct A_I from A_{ap} are extremely simple. Note also that Theorem 8 provides an alternate way to show that all polynomials can be defined by level automata. Finally, we illustrate Theorem 8 by the following example.

Example 6. Consider the automaton B of Example 5. As we saw $\hat{f}_A$ is continuous but does not possess derivatives at any point having a finite binary representation, nor at many other points as well. Since B is already average preserving, we can apply directly the rules (i)–(iv) and hence obtain the WFA B_I shown in Fig. 13. Of course, $\hat{f}_{B_I}$ is continuous, and $\hat{f}'_{B_I}(x) = \hat{f}_B(x)$. However, $\hat{f}_{B_I}$ does not have the second derivative at many points. Observe also that, as illustrated in Fig. 14,

$$\hat{f}_{B_I}\left(\frac{11}{32}\right) = \frac{1}{4}F_B(00) + \frac{1}{16}F_B(0100) + \frac{1}{32}F_B(01010).$$

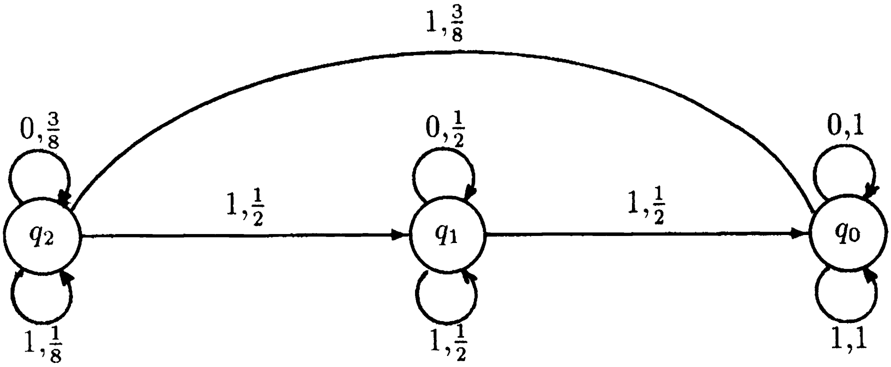


FIG. 13. *Automaton* B_I.

Note that by the integration (rules (i)–(iv)) starting from a noncontinuous function defined by a level automaton we obtain an automaton specifying a continuous function that, however, is not smooth.

7. Recent work. Since the first draft of this paper, the investigations of WFA continued intensively. We mention some recent results. In [4] it is shown that for each $n \ge 1$ the Daubechies wavelet W_{2n} [7] can be implemented by a WFA with $2n - 1 + \lceil \log(2n-1) \rceil$ states. These WFA are the first natural examples of WFA that are not level WFA. It is important that all the discrete W_{2n} transforms of size 2^k can be computed by a WFA with $2^k + 0(k)$ nodes,

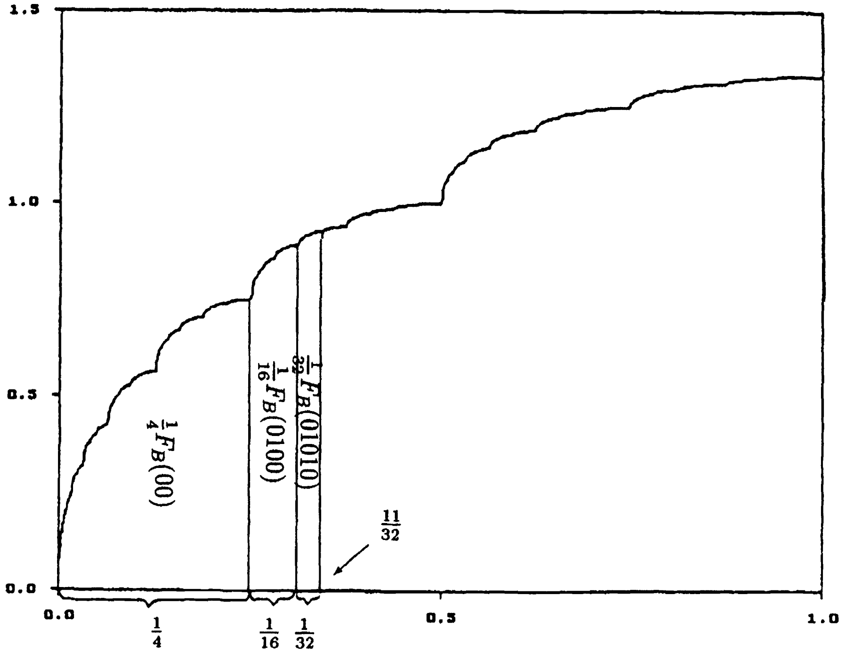


FIG. 14. *The graph of* $\hat{f}_B$.

a fixed structure and weights, and only the initial distribution depending on the given vector (function). This situation is similar to that of the polynomials of degree n considered in §5.

In [5] the first inference algorithm for WFA is given. Theoretically, for an infinitely precisely given function (i.e., function on [0, 1]), this algorithm produces a WFA with a minimal number of states that generates the given function if such a WFA exists. For a function f given by a finite table (finite resolution image), it produces a WFA that approximates f and has a relatively small number of states but possibly many edges (not very sparse weight matrices).

The second inference algorithm for the discrete case is described in [6]. It produces a WFA with highly sparse weight matrices, that is, a WFA with a relatively large number of states but a relatively small number of edges. This algorithm gives an excellent tool for data compression. For gray-scale images, it performs better than Daubechies W_6 wavelets and in combination with wavelets gives even higher compression rates with the same quality of the regenerated images. In this case the WFA can be viewed as a sophisticated method for the quantization of the wavelet coefficients [6].

Further theoretical results on WFA are shown in [8] and [9]. In [8] it is shown that the only smooth functions (functions that have everywhere all the derivatives) defined by WFA are the polynomials. In [9] a WFA is given that generates a function that is everywhere continuous but does not have the derivative anywhere.

Acknowledgment. The authors are grateful to J. Kari for useful discussions. The research reported here benefited from related work with him.

REFERENCES

[1] M. F. BARNSLEY, *Fractals Everywhere*, Academic Press, New York, 1988.

[2] J. BERSTEL AND CH. REUTENAUER, *Rational Series and Their Languages*, Springer-Verlag, Berlin, 1988.

[3] K. Culik ii and S. Dube, *Rational and affine expressions for image description*, Discrete Appl. Math., 41 (1993), pp. 85–120.

[4] ———, *Implementing wavelet transform with automata and application to data compression*, Proceedings of STACS '93, Lecture Notes in Comput. Sci., 665, Springer-Verlag, New York, 1993, pp. 343–353.

[5] K. Culik ii and J. Kari, *Image compression using weighted finite automata*, Computer and Graphics, 17 (1993), pp. 305–313.

[6] ———, *Image data compression using edge-optimizing algorithm for* WFA *inference*, J. Inform. Process. Management, to appear.

[7] I. Daubechies, *Orthonormal basis of compactly supported wavelets*, Comm. Pure Appl. Math., 41 (1988), pp. 909–996.

[8] D. Derencourt, J. Karhumäki, M. Latteux, and A. Terlutte, *On computational power of weighted finite automata*, Proceedings of MFCS 1992, Lecture Notes in Comput. Sci., 629, Springer-Verlag, New York, 1992, pp. 236–245.

[9] ———, *On Continuous Functions Computed by Finite Automata*, manuscript.

[10] S. Eilenberg, *Automata, Languages and Machines, Vol. A*, Academic Press, New York, 1974.

[11] F. Gesceg and I. Peck, *Algebraic Theory of Automata*, Akademiai Kiado, Budapest, 1972.

[12] A. Paz, *Introduction to Probabilistic Automata*, Academic Press, New York, 1971.

[13] A. Salomaa and M. Soittola, *Automata-Theoretic Aspects of Formal Power Series*, Springer-Verlag, Berlin, 1978.

[14] W. G. Tzeng, *The equivalence and learning of probabilistic automata*, The Proceedings of the 30th FOCS, 1989, pp. 268–273.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 815–833, August 1994

ON THE COMPLEXITY OF BILINEAR FORMS OVER ASSOCIATIVE ALGEBRAS*

NADER H. BSHOUTY†

Abstract. Let F be a field, and let $q(\alpha) = q_1^{d_1}(\alpha)\cdots q_k^{d_k}(\alpha) \in F[\alpha]$ be a polynomial of degree n, where $q_1(\alpha), \ldots, q_k(\alpha)$ are distinct irreducible polynomials. Let $y(\alpha), y_1(\alpha), \ldots, y_r(\alpha), x_1(\alpha), \ldots, x_s(\alpha)$ be $(n-1)$-degree polynomials with distinct nonscalar coefficients. The authors show the following: the number of nonscalar multiplications/divisions required to compute the coefficients of $x_i(\alpha)y(\alpha) \bmod q(\alpha)$ for $i = 1, \ldots, s$ by straight line algorithms is $s(2n-k)$. If H is a $s \times r$- matrix with entries from F, then the number of nonscalar multiplications/ divisions required to compute the coefficients of $(x_1(\alpha), \ldots, x_s(\alpha))\, H(y_1(\alpha), \ldots, y_r(\alpha))^T \bmod q(\alpha)$ by straight line algorithms is equal to $(2n-k)\mathrm{rank}(H)$. All the above systems satisfy the direct sum conjecture strongly. The above results also hold for some other algebras that are direct sums of local algebras, such as commutative algebras and division algebras.

Key words. associative algebras, bilinear forms, multiplicative complexity, direct sum

AMS subject classifications. 68C20, 68C25, 68Q05, 68Q25, 68Q40

1. Introduction.

Is it always true that the *multiplicative complexity* (i.e., the number of nonscalar multiplications) of multiplying one element x with many elements $y_1, \ldots, y_n$ in an algebra is equal to n times the multiplicative complexity of multiplying two elements in the algebra? The answer is "no." One simple example is $\mathcal{M}_{2\times 2}$, the algebra of 2×2 matrices: to multiply two 2×2 matrices we need seven nonscalar multiplications, and multiplying a 2×2 matrix X with n 2×2 matrices $Y_1, \ldots, Y_n$ requires $6n + 2$ nonscalar multiplications.

In [5] and [13], it is proved that the answer to the above question is "yes" when the algebra is a field. In this paper we study this property for other algebras and prove that it holds for commutative algebras of minimal rank. In particular, it holds for simply generated algebras (i.e., multiplication of two polynomials modulo a fixed polynomial). The technique used in this paper can also be used to prove this property for clean algebras of minimal rank. A clean algebra is an algebra $\mathcal{A}$ where $\mathcal{A}/\mathrm{rad}\,\mathcal{A}$ is a direct product of local algebras.

Let F be a field, and let $\mathbf{x} = (x_1, \ldots, x_n)^T$ be a vector of indeterminates . Let $Q_{\mathbf{x}} = \{\mathbf{x}^T Q_1 \mathbf{x}, \ldots, \mathbf{x}^T Q_m \mathbf{x}\}$ be a set of quadratic forms on $x_1, \ldots, x_n$ over F, where the Q_i are $n \times n$ matrices with entries from F. A *straight line algorithm* that computes $Q_{\mathbf{x}}$ is a sequence of rational functions $\sigma_1, \ldots, \sigma_L$, where:
(1) for every $1 \le j \le L$ we have $\sigma_j = w_{j,1} \circ w_{j,2}$, where $\circ \in \{\times, \div\}$ and

$$w_{j,1}, w_{j,2} \in \left(F + \sum_{i=1}^{n} Fx_i + \sum_{i=1}^{j-1} F\sigma_i\right) \backslash F$$

or $\circ = \div$, $w_{j,1} \in F\backslash\{0\}$, and $w_{j,2} \in \left(F + \sum_{i=1}^{n} Fx_i + \sum_{i=1}^{j-1} F\sigma_i\right) \backslash F$;
and
(2)

$$Q_{\mathbf{x}} \subseteq F + \sum_{i=1}^{n} Fx_i + \sum_{i=1}^{L} F\sigma_i.$$

* Received by the editors December 7, 1990; accepted for publication (in revised form) May 19, 1993. This work was supported by the NSERC of Canada.

† Department of Computer Science, Technion-Israel Institute of Technology, Haifa, Israel, and Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4 (bshouty@cpsc.ucalgary.ca).

We call the operation $\circ$ in (1) a *nonscalar* $\circ$. In this model we only count nonscalar multiplications/divisions. The minimal L in (1) is denoted by $L(Q_{\mathbf{x}})$ or $L_F(Q_{\mathbf{x}})$ and is called the *multiplicative complexity* of $Q_{\mathbf{x}}$.

When $Q_{\mathbf{x}}$ is computed by an algorithm $\sigma_1, \dots, \sigma_\mu$, where $\sigma_j = w_{j,1} \times w_{j,2}$ and $w_{j,1}, w_{j,2} \in \sum_{i=1}^{n} Fx_i \backslash F$, then we call the algorithm a *quadratic algorithm*. The minimal μ is denoted by $\mu(Q_{\mathbf{x}})$ or $\mu_F(Q_{\mathbf{x}})$ and is called the *quadratic complexity* of $Q_{\mathbf{x}}$. Strassen [17] proved that for infinite fields F,

$$L_F(Q_{\mathbf{x}}) = \mu_F(Q_{\mathbf{x}}). \tag{1}$$

Let $\mathbf{u} = (u_1, \dots, u_n)^T$ be a vector of new indeterminates, and let

$$Q_{\mathbf{x}}^{(1)} = \{\mathbf{x}^T Q_1 \mathbf{x}, \dots, \mathbf{x}^T Q_{m_1} \mathbf{x}\} \quad \text{and} \quad Q_{\mathbf{u}}^{(2)} = \{\mathbf{u}^T Q_{m_1+1} \mathbf{u}, \dots, \mathbf{u}^T Q_{m_2} \mathbf{u}\}$$

be two sets of quadratic forms. When the vectors of indeterminates $\mathbf{x}$ and $\mathbf{u}$ are distinct, $Q_{\mathbf{x}}^{(1)} \cup Q_{\mathbf{u}}^{(2)}$ is called the *direct sum* of $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$. It is also denoted by $Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)}$. It is obvious that

$$\mu(Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)}) \le \mu(Q_{\mathbf{x}}^{(1)}) + \mu(Q_{\mathbf{u}}^{(2)}). \tag{2}$$

Fiduccia and Zalcstein [9] Strassen [17], and Winograd [18] conjectured that for every two sets of quadratic forms $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$, we have

$$\mu(Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)}) = \mu(Q_{\mathbf{x}}^{(1)}) + \mu(Q_{\mathbf{u}}^{(2)}). \tag{3}$$

Furthermore, they conjectured that every minimal quadratic algorithm $\sigma_1, \dots, \sigma_\mu$ for $(Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)})$ can be split into two minimal algorithms

$$s_1 = (\sigma_i)_{i \in I} \quad \text{and} \quad s_2 = (\sigma_i)_{i \in J}, \tag{4}$$

where $I \cup J = \{1, \dots, \mu\}$, $I \cap J = \emptyset$, and s_1 and s_2 are minimal quadratic algorithms for $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$, respectively.

When (3) is satisfied, then we say that $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$ satisfy the *direct sum conjecture* in the model of quadratic algorithms. We define DSC_{QA} and $DSC_{QA}(F)$ to be the set of all pairs $(Q_{\mathbf{x}}^{(1)}, Q_{\mathbf{u}}^{(2)})$, such that $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$ satisfy the direct sum conjecture in the quadratic algorithms model. When (4) is satisfied for $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$, then, we say that $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$ satisfy the *direct sum conjecture strongly* in the model of quadratic algorithms. We define $DSCS_{QA}$ or $DSCS_{QA}(F)$ to be the set of all pairs $(Q_{\mathbf{x}}^{(1)}, Q_{\mathbf{u}}^{(2)})$ such that $Q_{\mathbf{x}}^{(1)}$ and $Q_{\mathbf{u}}^{(2)}$ satisfy the direct sum conjecture strongly in the quadratic algorithms model. If for any set of quadratic form $Q_{\mathbf{u}}^{(2)}$ we have that $(Q_{\mathbf{x}}^{(1)}, Q_{\mathbf{u}}^{(2)}) \in DSCS_{QA}$, then we simply write $Q_{\mathbf{x}}^{(1)} \in DSCS_{QA}$.

Similarly, we define the classes DSC_{SLA} and $DSCS_{SLA}$ for the straight line model. It is obvious that

$$DSCS_M \subseteq DSC_M$$

for any model of computation, M. By the results of Strassen in [17] and Bshouty in [8], for infinite fields F, we also have

$$DSC_{SLA}(F) = DSC_{QA}(F) \quad \text{and} \quad DSCS_{SLA}(F) = DSCS_{QA}(F). \tag{5}$$

Let $\mathbf{x} = (x_1, \dots, x_n)^T$ and $\mathbf{y} = (y_1, \dots, y_m)^T$ be vectors of indeterminates. Let $B_{\mathbf{x},\mathbf{y}}$ be a set of bilinear forms $\{\mathbf{x}^T B_1 \mathbf{y}, \dots, \mathbf{x}^T B_k \mathbf{y}\}$, where the B_i are $n \times m$ matrices. A *bilinear* *algorithm* that computes $B_{\mathbf{x},\mathbf{y}}$ is a quadratic algorithm $\sigma_1, \ldots, \sigma_\delta$, where $\sigma_j = w_{j,1} \times w_{j,2}$, $w_{j,1} \in \sum_{i=1}^{n} Fx_i \backslash F$ and $w_{j,2} \in \sum_{i=1}^{m} Fy_i \backslash F$. In a similar manner we define $\delta(B_{\mathbf{x},\mathbf{y}})$ to be the *bilinear complexity* of $B_{\mathbf{x},\mathbf{y}}$. Obviously, $\mu(B_{\mathbf{x},\mathbf{y}}) \leq \delta(B_{\mathbf{x},\mathbf{y}})$. We also define DSC_{BA} and $DSCS_{BA}$ in a similar manner.

Let $\mathbf{A}$ be an associative algebra of dimension k with an identity element 1, and let $\{a_1, \ldots, a_k\}$ be a basis of $\mathbf{A}$. Let $\mathbf{x} = (x_1, \ldots, x_n)^T$ and $\mathbf{y} = (y_1, \ldots, y_n)^T$ be vectors of indeterminates, and let $x = \sum_{i=1}^{k} x_i a_i$ and $y = \sum_{i=1}^{k} y_i a_i$. We denote by $[xy]_{\mathbf{A}} = \{\mathbf{x}^T B_1 \mathbf{y}, \ldots, \mathbf{x}^T B_k \mathbf{y}\}$, the set of bilinear forms defined by the product of two elements in the algebra $\mathbf{A}$. That is,

$$\sum_{i=1}^{k} (\mathbf{x}^T B_i \mathbf{y}) a_i = \left(\sum_{i=1}^{k} x_i a_i \right) \left(\sum_{i=1}^{k} y_i a_i \right).$$

In a similar manner we also define $[x_1 y, x_2 y, \ldots, x_n y]_{\mathbf{A}}$ and $[x_1 y_1 + \ldots + x_n y_n]_{\mathbf{A}}$, or more generally, $[x^T C_1 y, \ldots, x^T C_h y]_{\mathbf{A}}$, where $x = (x_1, \ldots, x_n)^T \in \mathbf{A}^n$, $y = (y_1, \ldots, y_m)^T \in \mathbf{A}^m$, and where $C_1, \ldots, C_h$ are $n \times m$ matrices. Formally, let $x_i = \sum_{j=1}^{k} x_{i,j} a_j$ and $y_l = \sum_{j=1}^{k} y_{l,j} a_j$ for $j = 1, \ldots, n$ and $l = 1, \ldots, m$, where $x_{i,j}$ and $y_{l,j}$ are distinct indeterminates. Let $x = (x_1, \ldots, x_n)^T$ and $y = (y_1, \ldots, y_m)^T$. Then $[x^T C_1 y, \ldots, x^T C_h y]_{\mathbf{A}}$ is the set of bilinear forms that is the set of the coefficients of $x^T C_i y$, $i = 1, \ldots, h$.

Obviously, $[x^T C_1 y, \ldots, x^T C_h y]_{\mathbf{A}}$ depends on the chosen basis $\{a_1, \ldots, a_k\}$. Feduccia and Zalcstein [9] proved that the complexity of $[x^T C_1 y, \ldots, x^T C_h y]_{\mathbf{A}}$ does not depend on the chosen basis.

In [4], Alder and Strassen proved that for any set of quadratic forms $Q_{\mathbf{u}}$ we have

$$\mu([xy]_{\mathbf{A}} \oplus Q_{\mathbf{u}}) \geq 2 \dim \mathbf{A} - I(\mathbf{A}) + \mu(Q_{\mathbf{u}}),$$

where $I(\mathbf{A})$ is the number of the maximal two sided ideals of $\mathbf{A}$. This result was generalized by Auslander and Winograd [5] and Hartman [13]. They proved that if $\mathbf{A}$ is a direct product of division algebras, then

$$\mu([x_1 y, x_2 y, \ldots, x_n y]_{\mathbf{A}}) \geq (2 \dim \mathbf{A} - I(\mathbf{A}))n.$$

If $\mu([x_1 y, x_2 y, \ldots, x_n y]_{\mathbf{A}}) \leq (2 \dim \mathbf{A} - I(\mathbf{A}))n$, then we call $\mathbf{A}$ an algebra of *n-minimal complexity*, and if $\delta([x_1 y, x_2 y, \ldots, x_n y]_{\mathbf{A}}) \leq (2 \dim \mathbf{A} - I(\mathbf{A}))n$, then we call $\mathbf{A}$ an algebra of *n-minimal rank*. Obviously, if $\mathbf{A}$ is an algebra of 1-minimal complexity (rank), then it is an algebra of n-minimal complexity (rank) for every n. In [10], De Groote proved that a division algebra $\mathbf{A}$ over a field F is an algebra of 1-minimal complexity if and only if $\mathbf{A}$ is a field with $|F| \geq 2 \dim \mathbf{A}$. Other characterizations of 1-minimal rank algebras over closed fields can be found in [12] and [15].

In this paper, we generalize all of the results in [4], [5], [6], [13], and [18]. We prove the following:

THEOREM I. *Let $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ be an algebra, where the $\mathbf{A}_i$ are local algebras and there exists some $d_i \in \mathbf{A}_i$ such that the left annihilator of the radical $l(\text{rad } \mathbf{A}_i) = \{a | a \text{ rad } \mathbf{A}_i = 0\} = d_i \mathbf{A}_i$. Then for any set of quadratic forms C, we have*

$$\mu([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus C) \geq (2 \dim \mathbf{A} - k)n + \mu(C).$$

If $\mathbf{A}$ is an algebra of n-minimal complexity, then

$$[x_1 y, \ldots, x_n y]_{\mathbf{A}} \in DSCS_{SLA}.$$

THEOREM II. *Let* $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ *be an algebra, where the* $\mathbf{A}_i$ *are local algebras. Then for any set of bilinear forms* B, *we have*

$$\delta([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus B) \geq (2 \dim \mathbf{A} - k)n + \delta(B).$$

If $\mathbf{A}$ *is an algebra of* n*-minimal rank, then*

$$[x_1 y, \ldots, x_n y]_{\mathbf{A}} \in DSCS_{BA}.$$

Notice that in Theorems I and II, $I(\mathbf{A}) = k$.

THEOREM III. *Let* $\mathbf{A}$ *be a commutative algebra, and let* H *be an* $n \times m$ *matrix. Then*

$$\mu([x^T H y]_{\mathbf{A}}) \geq (2 \dim \mathbf{A} - I(\mathbf{A}))\mathrm{rank}(H).$$

If $\mathbf{A}_1$ *and* $\mathbf{A}_2$ *are commutative algebras of* $\mathrm{rank}(H_1)$*-minimal rank and* $\mathrm{rank}(H_2)$*-minimal rank, respectively, then*

$$([x^T H_1 y]_{\mathbf{A}_1}, [x^T H_2 y]_{\mathbf{A}_2}) \in DSCS_{SLA}.$$

THEOREM IV. *Let* $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ *be an algebra, where the* $\mathbf{A}_i$ *are local algebras, and let* H *be an* $n \times m$ *matrix. Then for any set of bilinear forms* B, *we have*

$$\delta([x^T H y]_{\mathbf{A}} \oplus B) \geq (2 \dim \mathbf{A} - k)\mathrm{rank}(H) + \delta(B).$$

If $\mathbf{A}$ *is an algebra of* $\mathrm{rank}(H)$*-minimal rank, then*

$$[x^T H y]_{\mathbf{A}} \in DSCS_{BA}.$$

Since, by the Artinian theorem, every commutative algebra is a direct product of local commutative algebras, Theorems II and IV are true for commutative algebras. In this case we have $k = I(\mathbf{A})$.

Theorems I, II, III, and IV imply the following results:

Let $\mathbf{A} = F[\alpha]/(p(\alpha))$, where $p(\alpha) = p_1^{d_1}(\alpha) \ldots p_k^{d_k}(\alpha) \in F[\alpha]$ is a polynomial of degree n and $p_1, \ldots, p_k$ are distinct irreducible polynomials. Then $[xy]_{\mathbf{A}}$ is the problem of computing the product of two $(n-1)$-degree polynomials modulo $p(\alpha)$. By Theorems I and II, we have:

COROLLARY I. *Let* $\mathbf{A} = F[\alpha]/(p(\alpha))$, *where* $p = p_1^{d_1} \ldots p_k^{d_k}$, $p_1, \ldots, p_k$ *are distinct irreducible polynomials and* $|F| \geq 2 \max_{1 \leq i \leq k} \deg\ p_i^{d_i} - 2$. *Then for any set of quadratic forms* C *and any set of bilinear forms* B, *we have*

$$\mu([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus C) = (2 \deg\ p - k)n + \mu(C),$$

$$\delta([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus B) = (2 \deg\ p - k)n + \delta(B),$$

and

$$[x_1 y, \ldots, x_n y]_{\mathbf{A}} \in DSCS_{SLA}, DSCS_{BA}.$$

For a set of quadratic forms $Q_{\mathbf{x}}^{(1)}$ (respectively, bilinear forms $B_{\mathbf{x},\mathbf{y}}$), let $\mathrm{Alg}_{SLA}(Q_{\mathbf{x}}^{(1)})$ (respectively, $\mathrm{Alg}_{BA}(B_{\mathbf{x},\mathbf{y}})$) denote the set of all minimal straight line algorithms that compute $Q_{\mathbf{x}}^{(1)}$ (respectively, bilinear algorithms that compute $B_{\mathbf{x},\mathbf{y}}$). We denote by $\mathrm{Alg}_{SLA}(Q_{\mathbf{x}}^{(1)}) \oplus \mathrm{Alg}_{SLA}(Q_{\mathbf{u}}^{(2)})$, the set of all straight line algorithms $\sigma_1, \dots, \sigma_t$ such that there exist sets I, $J \subset \{1, \dots, t\}$, $I \cup J = \{1, \dots, t\}$ and $I \cap J = \emptyset$; where $(\sigma_i)_{i \in I} \in \mathrm{Alg}_{SLA}(Q_{\mathbf{x}}^{(1)})$ and $(\sigma_i)_{i \in J} \in \mathrm{Alg}_{SLA}(Q_{\mathbf{u}}^{(2)})$. Obviously, $(Q_{\mathbf{x}}^{(1)}, Q_{\mathbf{u}}^{(2)}) \in DSCS_{SLA}$ if and only if $\mathrm{Alg}_{SLA}(Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)}) = \mathrm{Alg}_{SLA}(Q_{\mathbf{x}}^{(1)}) \oplus \mathrm{Alg}_{SLA}(Q_{\mathbf{u}}^{(2)})$. By Corollary I, we have:

COROLLARY II. *Let* $\mathbf{A} = F[\alpha]/(p(\alpha))$, *where* $p = p_1^{d_1} \dots p_k^{d_k}$, $p_1, \dots, p_k$ *are distinct irreducible polynomials and* $|F| \geq 2\max_{1 \leq i \leq k} \deg\ p_i^{d_i} - 2$. *Then for the basis* $\{a_1, \dots, a_n\}$ *that represents* $\mathbf{A}$ *as* $F[\alpha]/(p_1^{d_1}(\alpha)) \times \cdots \times F[\alpha]/(p_k^{d_k}(\alpha))$, *we have*

$$\begin{aligned}\mathrm{Alg}_{SLA}([x_1 y, \dots, x_n y]_{\mathbf{A}}) = {} & \mathrm{Alg}_{SLA}([x_1 y, \dots, x_n y]_{F[\alpha]/(p_1^{d_1}(\alpha))}) \oplus \cdots \\ & \oplus \mathrm{Alg}_{SLA}([x_1 y, \dots, x_n y]_{F[\alpha]/(p_k^{d_k}(\alpha))})\end{aligned}$$

and

$$\begin{aligned}\mathrm{Alg}_{BA}([x_1 y, \dots, x_n y]_{\mathbf{A}}) = {} & \mathrm{Alg}_{BA}([x_1 y, \dots, x_n y]_{F[\alpha]/(p_1^{d_1}(\alpha))}) \oplus \cdots \\ & \oplus \mathrm{Alg}_{BA}([x_1 y, \dots, x_n y]_{F[\alpha]/(p_k^{d_k}(\alpha))}).\end{aligned}$$

This corollary shows that a classification of all the minimal straight line algorithms (respectively, bilinear algorithms) for $[x_1 y, \dots, x_n y]_{\mathbf{A}}$, where $\mathbf{A} = F[\alpha]/(p_i^{d_i}(\alpha))$ in which $p_i(\alpha)$ is irreducible, gives a classification for all the minimal straight line algorithms (respectively, bilinear algorithms) for $[x_1 y, \dots, x_n y]_{\mathbf{A}}$, where $\mathbf{A} = F[\alpha]/(p(\alpha))$, for *any* polynomial $p(\alpha)$. A classification of all minimal bilinear algorithms for the case of $n = 1$ is completely studied in [2] and [3].

Theorems III and IV imply:

COROLLARY III. *Let* $\mathbf{A} = F[\alpha]/(p(\alpha))$, *where* $p = p_1^{d_1} \dots p_k^{d_k}$, $p_1, \dots, p_k$ *are distinct irreducible polynomials and* $|F| \geq 2\max_{1 \leq i \leq k} \deg\ p_i^{d_i} - 2$. *Then*

$$\mu([x^T H y]_{\mathbf{A}}) = (2 \deg\ p(\alpha) - k)\mathrm{rank}(H),$$

and for any set of bilinear forms B,

$$\delta([x^T H y]_{\mathbf{A}} \oplus B) = (2 \deg\ p(\alpha) - k)\mathrm{rank}(H) + \delta(B)$$

and

$$[x^T H y]_{\mathbf{A}} \in DSCS_{BA}.$$

As in Corollary II, we have

COROLLARY IV. *Let* $\mathbf{A} = F[\alpha]/(p(\alpha))$, *where* $p = p_1^{d_1} \dots p_k^{d_k}$, $p_1, \dots, p_k$ *are distinct irreducible polynomials and* $|F| \geq 2\max_{1 \leq i \leq k} \deg\ p_i^{d_i} - 2$. *Then for the basis* $\{a_1, \dots, a_n\}$ *that represents* $\mathbf{A}$ *as* $F[\alpha]/(p_1^{d_1}(\alpha)) \times \cdots \times F[\alpha]/(p_k^{d_k}(\alpha))$, *we have*

$$\mathrm{Alg}_{SLA}([x^T H y]_{\mathbf{A}}) = \mathrm{Alg}_{SLA}([x^T H y]_{F[\alpha]/(p_1^{d_1}(\alpha))}) \oplus \cdots \oplus \mathrm{Alg}_{SLA}([x^T H y]_{F[\alpha]/(p_k^{d_k}(\alpha))})$$

and

$$\mathrm{Alg}_{BA}([x^T H y]_{\mathbf{A}}) = \mathrm{Alg}_{BA}([x^T H y]_{F[\alpha]/(p_1^{d_1}(\alpha))}) \oplus \cdots \oplus \mathrm{Alg}_{BA}([x^T H y]_{F[\alpha]/(p_k^{d_k}(\alpha))}).$$

The technique we shall use in this paper can also be used to prove lower bounds for many other bilinear systems over associative algebras.

This paper is organized as follows: In §2 we give some preliminary results from bilinear and quadratic complexity theory and the theory of associative algebras. In §§3 and 4 we study the complexity of $[x_1 y, \ldots, x_n y]_{\mathbf{A}}$ and $[x^T H y]_{\mathbf{A}}$, respectively.

2. Preliminary results. In this section we give some preliminary results.

2.1. Dual sets and equivalent sets.

DEFINITION 1. *Let $B = \{B_1, \ldots, B_k\}$ be a set of $n \times m$ matrices. We define the T-dual and D-dual sets of B to be*

$$B^T = \{B_1^T, \ldots, B_k^T\} \quad \text{and} \quad B^D = \{C_1, \ldots, C_m\},$$

respectively. Here B_i^T is the transpose of B_i and B^D denotes the set of $n \times k$ matrices that satisfy

$$C_i^{(j)} = B_j^{(i)}, \qquad i = 1, \ldots, m\ ;\ j = 1, \ldots, k,$$

where $B_i^{(j)}$ is the jth column of B_i. That is,

$$C_i = [B_1 e_{m,i} | \ldots | B_k e_{m,i}],$$

where $e_{m,i}$ is the ith column unit vector of order m.

We also define $B^E = B^{T\,DT}$, i.e., $B^E = \{D_1, \ldots, D_n\}$, where each

$$D_i = \begin{bmatrix} e_{n,i}^T B_1 \\ \vdots \\ e_{n,i}^T B_k \end{bmatrix}.$$

DEFINITION 2. *Let $B = \{B_1, \ldots, B_k\}$ be a set of $n \times m$ matrices. Let M, N, and $K = (K_{i,j})$ be $m \times m'$, $n' \times n$, and $k' \times k$ matrices, respectively. We define*

$$NBM = \{NB_1M, \ldots, NB_kM\} \quad \text{and} \quad B[K] = \left\{ \sum_{j=1}^{k} K_{1,j} B_j, \ldots, \sum_{j=1}^{k} K_{k',j} B_j \right\}.$$

DEFINITION 3. *Let $B = \{B_1, \ldots, B_k\}$ and $C = \{C_1, \ldots, C_{k'}\}$ be sets of $n \times m$ and $n' \times m'$ matrices, respectively. We define*

$$B \oplus C = \{\tilde{B}_1, \ldots, \tilde{B}_k, \tilde{C}_1, \ldots, \tilde{C}_{k'}\},$$

where

$$\tilde{B}_i = \begin{pmatrix} B_i & 0_{n \times m'} \\ 0_{n' \times m} & 0_{n' \times m'} \end{pmatrix}, \qquad \tilde{C}_j = \begin{pmatrix} 0_{n \times m} & 0_{n \times m'} \\ 0_{n' \times m} & C_j \end{pmatrix},$$

in which $0_{s \times r}$ denotes the zero $s \times r$ matrix.

Define

$$B \otimes C = \left\{ B_i \otimes C_j \middle|\ i = 1, \ldots, k\ ;\ j = 1, \ldots, k' \right\},$$

where $\otimes$ is the Kronecker product of matrices. If $k = k'$, then

$$\operatorname{diag}(B, C) = \{\operatorname{diag}(B_1, C_1), \ldots, \operatorname{diag}(B_k, C_k)\},$$

where

$$\operatorname{diag}(B_i, C_i) = \begin{pmatrix} B_i & 0_{n\times m'} \\ 0_{n'\times m} & C_i \end{pmatrix}.$$

The following properties, which were shown in [7], hold for these definitions:

LEMMA 1 ([7]). *Let A_1, A_2, and A be sets of k_1, k_2, and k matrices of size $n_1 \times m_1$, $n_2 \times m_2$, and $n \times m$, respectively. Let N, M, and K be matrices of appropriate size and I_r be the identity matrix of order r. Then:*

(1) $A^{TT} = A$, $A^{DD} = A$, $A^{EE} = A$, $A^E = A^{TDT} = A^{DTD}$.

(2) $A[K][J] = A[JK]$, $(NAM)[K] = N(A[K])M$.

(3) $(NAM)^T = M^T A^T N^T$, $(A[K])^T = A^T[K]$.

(4) $(NA)^D = NA^D$, $(AM)^D = A^D[M^T]$, $(A[K])^D = A^D K^T$.

(5) $(A_1 \oplus A_2)^T = A_1^T \oplus A_2^T$, $(A_1 \oplus A_2)^D = A_1^D \oplus A_2^D$.

(6) $(A_1 \otimes A_2)^T = A_1^T \otimes A_2^T$, $(A_1 \otimes A_2)^D = A_1^D \otimes A_2^D$.

(7) $A_1[K] \oplus A_2 = (A_1 \oplus A_2)[\operatorname{diag}(K, I_{k_2})]$, $NA_1M \oplus A_2 = \operatorname{diag}(N, I_{n_2})(A_1 \oplus A_2)\operatorname{diag}(M, I_{m_2})$.

(8) $A_1[K] \otimes A_2 = (A_1 \otimes A_2)[K \otimes I_{k_2}]$, $NA_1M \otimes A_2 = (N \otimes I_{n_2})(A_1 \otimes A_2)(M \otimes I_{m_2})$.

(9) $A \otimes (A_1 \oplus A_2) = (A \otimes A_1) \oplus (A \otimes A_2)$.

If we add $A^I = A$, then the set $\{I, T, D, TD, DT, E\}$ is a group that is isomorphic to the symmetric group $S_3 = \{(\,), (1, 2), (2, 3), (1, 3, 2), (1, 2, 3), (1, 3)\}$.

DEFINITION 4. *For two k-sets of $n \times m$ matrices B and C we say that B is equivalent to C and write $B \equiv C$, if there exist nonsingular matrices N, M, and K such that*

$$B = N(C[K])M.$$

Obviously, this relation is an equivalence relation.

The following lemma gives some properties of the equivalece relation.

LEMMA 2 ([7]). *Let $A_1, \ldots, A_j$; $B_1, \ldots, B_j$ be sets of matrices. Then:*

(1) *If $A_1 \equiv B_1$, then $A_1^D \equiv B_1^D$ and $A_1^T \equiv B_1^T$.*

(2) *$B_1 \oplus \cdots \oplus B_j \equiv B_{\phi(1)} \oplus \cdots \oplus B_{\phi(j)}$ and $B_1 \otimes \cdots \otimes B_j \equiv B_{\phi(1)} \otimes \cdots \otimes B_{\phi(j)}$ for any permutation ϕ on $\{1, \ldots, j\}$.*

(3) *If $A_i \equiv B_i$, $i = 1, \ldots, j$, then $A_1 \oplus \cdots \oplus A_j \equiv B_1 \oplus \cdots \oplus B_j$ and $A_1 \otimes \cdots \otimes A_j \equiv B_1 \otimes \cdots \otimes B_j$.*

2.2. Quadratic and bilinear complexity.

DEFINITION 5. *Let $A = \{A_1, \ldots, A_k\}$ be a set of matrices. We define*

$$\operatorname{rowrank}(A) = \operatorname{rank}[A_1 | \ldots | A_k], \qquad \operatorname{colrank}(A) = \operatorname{rank}[A_1^T | \ldots | A_k^T],$$

and

$$\dim(A) = \dim \operatorname{Span}(A),$$

where Span(A) *is the linear space spanned by the elements of A.*

The following can be easily proved

LEMMA 3. *Let A be a set of matrices. Then*

(1) rowrank$(A) = \dim(A^E)$, colrank$(A) = \dim(A^D)$.

(2) *If* $A \equiv B$, *then*

$$\text{rowrank}(B) = \text{rowrank}(A), \qquad \text{colrank}(B) = \text{colrank}(A).$$

Let $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$ be vectors of indeterminates. For a set of $n \times m$ matrices $A = \{A_1, \dots, A_k\}$, we denote by $\mu(A)$ (respectively, by $\delta(A)$) the quadratic complexity (respectively, bilinear complexity) of the bilinear forms $\{\mathbf{x}^T A_1 \mathbf{y}, \dots, \mathbf{x}^T A_k \mathbf{y}\}$. We will call $\{\mathbf{x}^T A_1 \mathbf{y}, \dots, \mathbf{x}^T A_k \mathbf{y}\}$ the set of bilinear forms *defined by* $A = \{A_1, \dots, A_k\}$.

From [8], [14], and [16] we have:

LEMMA 4. *Let A and B be sets of matrices. We have:*

(1) $\mu(A) \le \delta(A) < 2\mu(A)$.

(2) $\delta(A) = \delta(A^T) = \delta(A^D) = \delta(A^E)$, $\mu(A) = \mu(A^T)$.

(3) *If* $A \equiv B$, *then* $\mu(A) = \mu(B)$.

(4) $\mu(A \oplus B) \le \mu(A) + \mu(B)$ *and* $\mu(I_t \otimes A) \le \mu(\bigoplus_{i=1}^t A) \le t\mu(A)$.

(5) $\mu(A \otimes B) \le \mu(A)\mu(B)$.

(6) $\mu(A) \ge \frac{1}{2}\delta(\text{diag}(A, A^T))$.

(7) $DSC_{SLA} = DSC_{QA}$ *and* $DSCS_{SLA} = DSCS_{QA}$.

(8) *If* $A, B \in DSCS_{SLA}$, *then* $A \oplus B \in DSCS_{SLA}$.

(3), (4), (5), *and* (8) *are also true for the bilinear complexity*, δ.

From [6] we have:

LEMMA 5 ([6]). *Let A be a set of matrices. If for every nonsingular matrix N there exist* $A_1, \dots, A_s \in A[N]$ *such that*

$$\text{rowrank}\{A_1, \dots, A_s\} \ge t \qquad \text{or} \qquad \text{colrank}\{A_1, \cdots, A_s\} \ge t,$$

then

$$\mu(A) \ge \dim\ A + t - s.$$

2.3. Complexity of a direct sum of sets.

DEFINITION 6. *We denote by $DS^*(r)$ the collection of sets of matrices A, such that there exists a linear subspace A_1 of* Span(A) *and integers s and t, where:*

(1) *For any nonsingular matrix N and for any* $B_1 \in (\text{Span}(A)\backslash A_1) \cap A[N]$ *there exist* $s - 1$ *matrices* $B_2, \dots, B_s \in (\text{Span}(A)\backslash A_1) \cap A[N]$ *such that*

$$\text{rowrank}\{B_1, \dots, B_s\} \ge t \qquad \textit{or} \qquad \text{colrank}\{B_1, \dots, B_s\} \ge t,$$

and

(2) $\mu(A) = \dim(A) + t - s + r$.

In [6], we proved the following two lemmas.

LEMMA 6. (1) *If* $A \in DS^*(0)$, *then* $A \in DSCS_{SLA}$.

(2) *If* $A \in DS^*(1)$, *then* $A \in DSC_{SLA}$.

(3) *If* $A \in DS^*(r), r > 1$, *then for every set of matrices B we have*

$$\mu(A \oplus B) \ge \mu(A) + \mu(B) - (r - 1).$$

Notice that all the results in Lemmas 5 and 6 are also true for the bilinear complexity.

LEMMA 7 ([6]). $A \in DSCS_{BA}$ *if and only if* $A^D \in DSCS_{BA}$ *if and only if* $A^E \in DSCS_{BA}$.

Using the results in [16] we prove another lemma that will be used in this paper. For the proof, we need the following

PROPOSITION 1 ([6]). *Let* $\mathbf{x}$ *and* $\mathbf{u}$ *be distinct vectors of indeterminates and let* $Q_{\mathbf{x}}^{(1)}$ *and* $Q_{\mathbf{u}}^{(2)}$ *be two sets of quadratic forms. Then* $(Q_{\mathbf{x}}^{(1)}, Q_{\mathbf{u}}^{(2)}) \in DSCS_{SLA}$ *if and only if for any minimal quadratic algorithm* $\sigma_1, \dots, \sigma_t$ *for* $Q_{\mathbf{x}}^{(1)} \oplus Q_{\mathbf{u}}^{(2)}$ *we have* $\sigma_i \in F[\mathbf{x}] \cup F[\mathbf{u}]$ *for all* $1 \le i \le t$.

The proposition is also true for the bilinear complexity.

LEMMA 8. *Let* $A = \{A_1, \dots, A_k\}$ *and* $B = \{B_1, \dots, B_{k'}\}$ *be sets of matrices. If*

(1) $(\mathrm{diag}(A, A^T), \mathrm{diag}(B, B^T)) \in DSCS_{BA}$

and

(2) $\delta(\mathrm{diag}(A, A^T)) = 2\delta(A)$ *and* $\delta(\mathrm{diag}(B, B^T)) = 2\delta(B)$,

then $(A, B) \in DSCS_{SLA}$.

Proof. If $(A, B) \notin DSCS_{SLA}$, then by Proposition 1, there exists a minimal quadratic algorithm $\sigma_1, \dots, \sigma_t$ for $\{\mathbf{x}^T A_1 \mathbf{y}, \dots, \mathbf{x}^T A_k \mathbf{y}, \mathbf{u}^T B_1 \mathbf{v}, \dots, \mathbf{u}^T B_{k'} \mathbf{v}\}$, where $\sigma_{i_0} \in F[\mathbf{x}, \mathbf{y}, \mathbf{u}, \mathbf{v}] \setminus (F[\mathbf{x}, \mathbf{y}] \cup F[\mathbf{u}, \mathbf{v}])$ for some $1 \le i_0 \le t$.

Let

$$\sigma_i = (a_{i,1}(\mathbf{x}) + a_{i,2}(\mathbf{y}) + a_{i,3}(\mathbf{u}) + a_{i,4}(\mathbf{v}))(b_{i,1}(\mathbf{x}) + b_{i,2}(\mathbf{y}) + b_{i,3}(\mathbf{u}) + b_{i,4}(\mathbf{v})).$$

It is known from [16] that the algorithm $(\tilde{\sigma}_{i,j})_{i=1,\dots,t,\ j=1,2}$, where

$$\tilde{\sigma}_{i,1} = (a_{i,1}(\mathbf{x}) + a_{i,2}(\mathbf{x}') + a_{i,3}(\mathbf{u}) + a_{i,4}(\mathbf{u}'))(b_{i,1}(\mathbf{y}') + b_{i,2}(\mathbf{y}) + b_{i,3}(\mathbf{v}') + b_{i,4}(\mathbf{v}))$$

and

$$\tilde{\sigma}_{i,2} = (b_{i,1}(\mathbf{x}) + b_{i,2}(\mathbf{x}') + b_{i,3}(\mathbf{u}) + b_{i,4}(\mathbf{u}'))(a_{i,1}(\mathbf{y}') + a_{i,2}(\mathbf{y}) + a_{i,3}(\mathbf{v}') + a_{i,4}(\mathbf{v})),$$

is a bilinear algorithm that computes the bilinear forms defined by $D = \mathrm{diag}(A, A^T) \oplus \mathrm{diag}(B, B^T)$. Since

$$\delta(\mathrm{diag}(A, A^T) \oplus \mathrm{diag}(B, B^T)) = \delta(\mathrm{diag}(A, A^T)) + \delta(\mathrm{diag}(B, B^T)) = 2\delta(A) + 2\delta(B),$$

we have that $\{\tilde{\sigma}_{i,j}\}_{i=1,\dots,t,\ j=1,2}$ is a minimal bilinear algorithm for D. Since

$$(\mathrm{diag}(A, A^T), \mathrm{diag}(B, B^T)) \in DSCS_{BA}$$

and since

$$\tilde{\sigma}_{i_0,1} \in F[\mathbf{x}, \mathbf{x}', \mathbf{y}, \mathbf{y}', \mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{v}'] \setminus (F[\mathbf{x}, \mathbf{x}', \mathbf{y}, \mathbf{y}'] \cup F[\mathbf{u}, \mathbf{u}', \mathbf{v}, \mathbf{v}']),$$

we have a contradiction to Proposition 1, which implies the result. □

2.4. Regular representation of algebras. In this subsection we will give the results of [7] that will be used in the following sections.

Let **A** be an associative algebra with identity element 1, and let $\{a_1, \dots, a_n\}$ be a basis of the algebra **A**. Let

$$a_i a_j = \sum_{k=1}^{n} \gamma_{i,j,k} a_k,$$

with $\gamma_{i,j,k} \in F$, $i, j, k = 1, \dots, n$. Then for $x = \sum_{i=1}^n x_i a_i$ and $y = \sum_{j=1}^n y_j a_j$ we have

$$xy = \left(\sum_{i=1}^{n} x_i a_i\right)\left(\sum_{j=1}^{n} y_j a_j\right) = \sum_{k=1}^{n}\left(\sum_{i=1}^{n}\sum_{j=1}^{n} \gamma_{i,j,k} x_i y_j\right) a_k.$$

Let $a_i a = \sum_{k=1}^{n} \sigma_{i,k} a_k$, and define $A_a = (\sigma_{i,k})$ to be a square $n \times n$ matrix. Then it is known that $\mathbf{RR}(\mathbf{A}) = \{A_a | a \in \mathbf{A}\}$ forms an algebra over F that is isomorphic to $\mathbf{A}$ under the correspondence $a \to A_a$ and that $\{A_{a_1}, \ldots, A_{a_n}\}$ is a basis for the algebra $\mathbf{RR}(\mathbf{A})$. Therefore, we have: $A_\lambda = \lambda I_n$; $A_a A_b = A_{ab}$; $A_a + A_b = A_{a+b}$; $\lambda A_a = A_{\lambda a}$ for $\lambda \in F$; and if $ab = 1$, then $A_a^{-1} = A_b$. The algebra $\mathbf{RR}(\mathbf{A})$ is called the *right regular representation* of $\mathbf{A}$.

We define

$$\mathbf{B}(\mathbf{A}) = \{B_1, \ldots, B_n\},$$

where

$$\mathbf{x}^T B_i \mathbf{y} = \sum_{i=1}^{n} \sum_{j=1}^{n} \gamma_{i,j,k} x_i y_j,$$

for $\mathbf{x} = (x_1, \ldots, x_n)^T$, $\mathbf{y} = (y_1, \ldots, y_n)^T$. That is, $\mathbf{x}^T B_i \mathbf{y}$ is the ith coefficient of the product

$$xy = \left(\sum_{i=1}^{k} x_i a_i\right)\left(\sum_{i=1}^{k} y_i a_i\right).$$

Let $\mathbf{C}(\mathbf{A}) = \{A_{a_1}, \ldots, A_{a_n}\}$. In [7] we gave the following connection between $\mathbf{B}(\mathbf{A})$ and $\mathbf{C}(\mathbf{A})$.

LEMMA 9. *We have*

$$\mathbf{C}(\mathbf{A})^D = \mathbf{B}(\mathbf{A}).$$

Obviously, $\mathbf{C}(\mathbf{A})$ and $\mathbf{B}(\mathbf{A})$ depend on the chosen basis $B = \{a_1, \ldots, a_n\}$. When we want to emphasize this dependency, we write $\mathbf{C}(\mathbf{A}, B)$ and $\mathbf{B}(\mathbf{A}, B)$.

The following two lemmas are also well known.

LEMMA 10. *Let* $\mathbf{A}_1$ *and* $\mathbf{A}_2$ *be algebras. Let* $\mathbf{A}_1 \otimes \mathbf{A}_2$ *be the tensor product of the algebras* $\mathbf{A}_1$ *and* $\mathbf{A}_2$, *and let* $\mathbf{A}_1 \times \mathbf{A}_2$ *be the direct product of* $\mathbf{A}_1$ *and* $\mathbf{A}_2$. *Then*

$$\mathbf{C}(\mathbf{A}_1 \times \mathbf{A}_2) \equiv \mathbf{C}(\mathbf{A}_1) \oplus \mathbf{C}(\mathbf{A}_2), \quad \mathbf{B}(\mathbf{A}_1 \times \mathbf{A}_2) \equiv \mathbf{B}(\mathbf{A}_1) \oplus \mathbf{B}(\mathbf{A}_2),$$

$$\mathbf{C}(\mathbf{A}_1 \otimes \mathbf{A}_2) \equiv \mathbf{C}(\mathbf{A}_1) \otimes \mathbf{C}(\mathbf{A}_2), \quad \mathbf{B}(\mathbf{A}_1 \otimes \mathbf{A}_2) \equiv \mathbf{B}(\mathbf{A}_1) \otimes \mathbf{B}(\mathbf{A}_2).$$

For the next lemma we need the following

DEFINITION 7. *An algebra* $\mathbf{A}$ *is called a Frobenuis algebra if there exists a linear function* $\lambda \in \mathbf{A}^*$ *whose kernel contain no left or right ideals different from zero. For other definitions of Frobenuis algebras see* [19]. *The following lemma can also be a definition for Frobenuis algebras*

LEMMA 11. *Let* $\mathbf{A}$ *be an algebra, and let* $\mathbf{A}^-$ *be the reciprocal algebra. Then* $\mathbf{A}$ *is a Frobenuis algebra if and only if*

$$\mathbf{C}(\mathbf{A}^-) \equiv \mathbf{B}(\mathbf{A}).$$

In [7] it is proved that

LEMMA 12. *A local algebra* $\mathbf{A}$ *is a Frobenuis algebra if and only if there exists an element* $d \in \mathbf{A}$ *such that*

$$l(\text{rad}\mathbf{A}) = \{a | a \ \text{rad}\mathbf{A} = 0\} = d\mathbf{A}.$$

Let $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_n$ *where each* $\mathbf{A}_i$ *is a local algebra. Then* $\mathbf{A}$ *is a Frobenuis algebra if and only if each* $\mathbf{A}_i$ *is a Frobenuis algebra.*

Before we leave this subsection we shall prove the following.

LEMMA 13. *Let* $\mathbf{A}$ *be a local algebra. There exists a basis* $\{a_1, \ldots, a_k, a_{k+1}, \ldots, a_{rk}\}$ *for* $\mathbf{A}$*, such that* $a_1, \ldots, a_k \notin \text{rad } \mathbf{A}$, $a_{k+1}, \ldots, a_{rk} \in \text{rad } \mathbf{A}$,

$$(6) \qquad A_{a_i} = \begin{pmatrix} \tilde{A}_i & & * \\ & \ddots & \\ 0 & & \tilde{A}_i \end{pmatrix}, \qquad i = 1, \ldots, k,$$

where $\mathbf{C}(\mathbf{A}/\text{rad } \mathbf{A}) = \{\tilde{A}_i\}_{i=1,\ldots,k}$, *and*

$$(7) \qquad A_{a_i} = \begin{pmatrix} 0_{k\times k} & & * \\ & \ddots & \\ 0 & & 0_{k\times k} \end{pmatrix}, \qquad i = k+1, \ldots, rk.$$

Proof. Let $\mathbf{A} = L \oplus_F \text{rad } \mathbf{A}$, where L is a complementary subspace of rad $\mathbf{A}$ and $\oplus_F$ is the direct sum of subspaces. Let $\{a_1, \ldots, a_k\}$ be a basis for L. Since $\mathbf{A}$ is local, all the elements in L are invertible. Let i_0 be the least integer such that $(\text{rad } \mathbf{A})^{i_0} \neq 0$ and $(\text{rad } \mathbf{A})^{i_0+1} = 0$. We now prove that there exist $b_1, b_2, \ldots, b_{k_0} \in (\text{rad } \mathbf{A})^{i_0}$, such that $b_1 L \oplus_F \cdots \oplus_F b_{k_0} L = (\text{rad } \mathbf{A})^{i_0}$. We first choose an element $b_1 \in (\text{rad } \mathbf{A})^{i_0}$. Since L contains invertible elements, we have that $b_1 L \subseteq (\text{rad } \mathbf{A})^{i_0}$. Suppose $b_1 L \oplus_F \cdots \oplus_F b_k L \subseteq (\text{rad } \mathbf{A})^{i_0}$ for some $b_1, \ldots, b_k \in (\text{rad } \mathbf{A})^{i_0}$. Let $b_{k+1} \in (\text{rad } \mathbf{A})^{i_0} \backslash (b_1 L \oplus_F \cdots \oplus_F b_k L)$. Then $b_{k+1} L \subseteq (\text{rad } \mathbf{A})^{i_0}$. If $b_{k+1} L \cap (b_1 L \oplus_F \cdots \oplus b_k L) \neq 0$, then there exist $a_1, \ldots, a_{k+1} \in L$ such that $b_{k+1} a_{k+1} = b_1 a_1 + \cdots + b_k a_k$, which implies that $b_{k+1} = b_1 a_1 a_{k+1}^{-1} + \cdots + b_k a_k a_{k+1}^{-1} \in b_1 \mathbf{A} + \cdots + b_k \mathbf{A} = b_1 (L \oplus_F \text{rad } \mathbf{A}) + \cdots + b_k (L \oplus_F \text{rad } \mathbf{A}) = b_1 L + \cdots + b_k L$. This is a contradiction. Therefore $b_1 L \oplus_F \cdots \oplus_F b_{k+1} L \subseteq (\text{rad } \mathbf{A})^{i_0}$.

This proves that there exist $b_1, \ldots, b_{k_0} \in (\text{rad } \mathbf{A})^{i_0}$ such that $\{b_i a_j\}_{\substack{i=1,\ldots,k_0 \\ j=1,\ldots,k}}$ is a basis for $(\text{rad } \mathbf{A})^{i_0}$ and

$$L_{i_0} = b_1 L \oplus_F \cdots \oplus_F b_{k_0} L = (\text{rad } \mathbf{A})^{i_0}.$$

We now handle the set $(\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}$. Let $b_{k_0+1} \in (\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}$. Then $b_{k_0+1} L \subseteq (\text{rad } \mathbf{A})^{i_0-1} \backslash\ (\text{rad } \mathbf{A})^{i_0}$ and therefore $b_{k_0+1} L \cap L_{i_0} = 0$. Suppose $b_{k_0+1} L \oplus_F \cdots \oplus_F b_k L \subseteq (\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}$. Let $b_{k_0+k+1} \in ((\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}) \backslash\ (b_{k_0+1} L \oplus_F \cdots \oplus_F b_k L)$, where $b_{k_0+1}, \ldots, b_k \in (\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}$. If $b_{k_0+k+1} L \cap (b_{k_0+1} L \oplus_F \cdots \oplus_F b_k L \oplus_F L_{i_0}) \neq \{0\}$, then there exist $a_1, \ldots, a_{k+1} \in L$ and $c \in L_{i_0}$, such that $b_{k_0+k+1} a_{k+1} = b_{k_0+1} a_1 + \cdots + b_{k_0+k} a_k + c$. Then $b_{k_0+k+1} = b_{k_0+1} a_1 a_{k+1}^{-1} + \cdots + b_{k_0+k} a_k a_{k+1}^{-1} + c a_{k+1}^{-1}$. Since the $a_i a_{k+1}^{-1}$ are invertible, we have $a_i a_{k+1}^{-1} = c_i + w_i$, where $0 \neq c_i \in L$ and $w_i \in \text{rad } \mathbf{A}$. As a result, $b_{k_0+k+1} = \sum_{i=1}^{k} b_{k_0+i} c_i + \sum_{i=1}^{k} b_{k_0+i} w_i + c a_{k+1}^{-1}$. Now, $\sum_{i=1}^{k} b_{k_0+1} w_i + c a_{k+1}^{-1} \in (\text{rad } \mathbf{A})^{i_0}$ and $c_i \in L$. Therefore $b_{k_0+k+1} \in b_{k_0+1} L + \cdots + b_{k_0+k-1} L + L_{i_0}$, which is a contradiction. In this way we can find $b_{k_0+1}, \ldots, b_{k_1}$ such that

$$L_{i_0-1} = b_{k_0+1} L \oplus_F \cdots \oplus_F b_{k_1} L = (\text{rad } \mathbf{A})^{i_0-1} \backslash (\text{rad } \mathbf{A})^{i_0}.$$

Therefore (by induction) there exist elements $b_1, \ldots, b_r$ such that

$$\mathbf{A} = L \oplus_F b_1 L \oplus_F \cdots \oplus_F b_r L$$

and such that if $b_{i_1} \in (\text{rad } \mathbf{A})^{j_1} \backslash (\text{rad } \mathbf{A})^{j_1-1}$, $b_{i_2} \in (\text{rad } \mathbf{A})^{j_2} \backslash (\text{rad } \mathbf{A})^{j_2-1}$ and $i_2 > i_1$, then $j_2 \leq j_1$.

We now find the regular representation $\mathbf{C}(\mathbf{A}, B)$, where $B = \{b_{r-i+1}a_j\}_{\substack{i=0,\ldots,r\\ j=1,\ldots,k}}$ and $b_0 = 1$.

Let $\phi : \mathbf{A} \to \mathbf{A}/\text{rad } \mathbf{A}$ be a canonical projection. Then $c_i = \phi(a_i)$, $i = 1, \ldots, k$, is a basis of $\mathbf{A}/\text{rad } \mathbf{A}$. If

$$c_i c_j = \sum_{l=1}^{k} \gamma_{i,j,l} c_l, \tag{8}$$

then since $\phi(a_i a_j) = c_i c_j = \sum_{l=1}^{k} \gamma_{i,j,l} c_l = \phi\left(\sum_{l=1}^{k} \gamma_{i,j,l} a_l\right)$, we have

$$a_i a_j = \sum_{l=1}^{k} \gamma_{i,j,l} a_l + w_{i,j}, \quad \text{for some } w_{i,j} \in \text{rad } \mathbf{A}.$$

Suppose

$$b_i \in (\text{rad } \mathbf{A})^{n_{b_i}} \backslash (\text{rad } \mathbf{A})^{n_{b_i}+1}, \qquad n_{b_i} \geq 1 \text{ and } b_0 = 1 \notin \text{rad } \mathbf{A}.$$

Now since

$$(b_t a_i) a_j = \sum_{l=1}^{k} \gamma_{i,j,l} b_t a_l + b_t w_{i,j}$$

and since $b_t w_{i,j} \in (\text{rad } \mathbf{A})^{n'} \backslash (\text{rad } \mathbf{A})^{n'+1}$ for $n' > n_{b_t}$, by (8) we have that A_{a_j} is a matrix with the shape in (6), where $\tilde{A}_j = A_{c_j}$ in $\mathbf{C}(\mathbf{A}/\text{rad } \mathbf{A}, \{c_1, \ldots, c_k\})$.

Since we have $(b_r a_i)(b_s a_j) \in b_{e-1}L \oplus_F \cdots \oplus_F b_1 L$ for $r, s > 0$, $e = \min(r, s)$, we also have that $A_{b_s a_j}$ is a matrix with the shape in (7). □

2.5. Complexity of algebras. Let $C = \{C_1, \ldots, C_k\}$ be a set of $n \times m$ matrices and let $\mathbf{A}$ be an algebra. The *quadratic complexity of C over the algebra* $\mathbf{A}$ is $\mu(C \otimes \mathbf{B}(\mathbf{A}))$, i.e., the complexity of $\{\mathbf{x}^T C_1 \mathbf{y}, \ldots, \mathbf{x}^T C_k \mathbf{y}\}$, where $\mathbf{x} = (x_1, \ldots, x_n)^T$, $\mathbf{y} = (y_1, \ldots, y_m)^T$ and x_i, y_j are elements in the algebra $\mathbf{A}$. In the same manner, we can define the bilinear complexity of C over the algebra $\mathbf{A}$ as $\delta(C \otimes \mathbf{B}(\mathbf{A}))$.

The following is proved in [7].

LEMMA 14. (1) $\delta(C \otimes \mathbf{B}(\mathbf{A})) = \delta(C^D \otimes \mathbf{C}(\mathbf{A}))$.

(2) $\mu(C \otimes \mathbf{B}(\mathbf{A})) = \mu(C^T \otimes \mathbf{B}(\mathbf{A}^-))$.

(3) *If* $\mathbf{A}$ *is a Frobenuis algebra, then* $\mu(C \otimes \mathbf{B}(\mathbf{A})) = \mu(C \otimes \mathbf{C}(\mathbf{A}))$.

3. Complexity of $x_1 y, \ldots, x_n y$ in algebras. In this section we investigate the complexity of computing $\{x_1 y, \ldots, x_n y\}$ in algebras that are direct product of local algebras. This problem is equivalent to the complexity of $\mathbf{I}_n^D = \{e_{n,1}, \ldots, e_{n,n}\}$ over the algebra $\mathbf{A}$, where $e_{n,i}$ is the ith column unit vector of order n and $\mathbf{I}_n = \{I_n\}$ in which I_n is the identity $n \times n$ matrix. Therefore,

$$\mu([x_1 y, \ldots, x_n y]) = \mu(\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A})).$$

Since $(\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}))^D = \mathbf{I}_n \otimes \mathbf{C}(\mathbf{A}) = \left\{I_n \otimes A_{a_i} \middle| \{a_i\}_{i=1,\ldots,k} \text{ is a basis for } A\right\}$, we have

$$\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) = \left\{[(I_n \otimes A_{a_1})e_{nk,i} \mid \ldots |(I_n \otimes A_{a_n})e_{nk,i}]\middle| i = 1, \ldots, nk\right\},$$

where $k = \dim\ \mathbf{A}$.

For $v \in F^{nk}$, we denote by $B(v)$ the matrix

$$B(v) = \left[(I_n \otimes A_{a_1})v\middle| \ldots \middle|(I_n \otimes A_{a_k})v\right].$$

Since $\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) = \{B(e_{nk,1}), \ldots, B(e_{nk,nk})\}$ and

$$\lambda_1 B(v_1) + \lambda_2 B(v_2) = B(\lambda_1 v_1 + \lambda_2 v_2), \tag{9}$$

we have that

$$\operatorname{Span}(\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A})) = \{B(v) | v \in F^{nk}\}.$$

We will first investigate the complexity of division algebras.

LEMMA 15. *Let* $\mathbf{A}$ *be a division algebra of dimension* k, *and let* $\{a_1, \ldots, a_k\}$ *be a basis for* $\mathbf{A}$. *Let* $v_1, \ldots, v_l \in F^{nk} \backslash \{0\}$ *and*

$$s = \dim \operatorname{Span}\left\{(I_n \otimes A_{a_i})v_j \middle| i = 1, \ldots, k;\ j = 1, \ldots, l\right\}.$$

Then for any nonsingular $l \times l$ *matrix* N, *there exist* $\frac{s}{k} - 1$ *matrices* $B(w_{i_2}), \ldots, B(w_{i_{(s/k)}})$ *in* $\{B(w_1), \ldots, B(w_l)\} = \{B(v_1), \ldots, B(v_l)\}[N]$, *such that*

$$\operatorname{rowrank}(\{B(w_1), B(w_{i_2}), \ldots, B(w_{i_{s/k}})\}) = s.$$

Proof. By (9) we have that $\{w_1, \ldots, w_l\} = \{v_1, \ldots, v_l\}[N] \subset F^{nk}$. Since N is nonsingular,

$$\operatorname{Span}(w_1, \ldots, w_l) = \operatorname{Span}(v_1, \ldots, v_l),$$

and therefore

$$\dim \operatorname{Span}\left\{(I_n \otimes A_{a_i})w_j \middle| i = 1, \ldots, k;\ j = 1, \ldots, l\right\} = s. \tag{10}$$

If $l = s/k$, then

$$\operatorname{rowrank}(\{B(w_1), B(w_2) \ldots, B(w_l)\}) = \operatorname{rank}[B(w_1)|B(w_2)| \ldots |B(w_l)]$$

$$= \dim \operatorname{Span}\left\{(I_n \otimes A_{a_i})w_j \middle| i = 1, \ldots, k,\ j = 1, \ldots, l\right\} = s, \tag{11}$$

and the proof is completed.

If $l > s/k$, then the set $\{(I_n \otimes A_{a_i})w_j | i = 1, \ldots, k\ ;\ j = 1, \ldots, l\}$ contains $lk > s$ vectors, and therefore there exists one vector that is dependent on the other. Assume that

$$\sum_{i=1}^{k} \lambda_i (I_n \otimes A_{a_i}) w_{j_0} = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq j_0}}^{l} \delta_{i,j} (I_n \otimes A_{a_i}) w_j \tag{12}$$

for some $1 \leq j_0 \leq l$, where not all the λ_i are zero. If all the $\delta_{i,j}$ are zero, then $\sum_{i=1}^{k} \lambda_i (I_n \otimes A_{a_i}) w_{j_0} = 0$, and therefore

$$\sum_{i=1}^{k} \lambda_i (I_n \otimes A_{a_i}) w_{j_0} = \left(I_n \otimes \left(\sum_{i=1}^{k} \lambda_i A_{a_i}\right)\right) w_{j_0} = (I_n \otimes A_a) w_{j_0} = 0, \tag{13}$$

for $a = \sum_{i=1}^{k} \lambda_i a_i \neq 0$. Since $\mathbf{A}$ is a division algebra, $I_n \otimes A_a$ is a nonsingular matrix and (13) implies that $w_{j_0} = 0$. This is a contradiction. Therefore, not all of the $\delta_{i,j}$ are zero. This implies that we can choose $j_0 \neq 1$, such that (12) is satisfied. Then for $a = \sum_{i=1}^{k} \lambda_i a_i$, we obtain

$$(I_n \otimes A_a)w_{j_0} = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq j_0}}^{l} \delta_{i,j}(I_n \otimes A_{a_i})w_j,$$

which implies

$$w_{j_0} = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq j_0}}^{l} \delta_{i,j}(I_n \otimes A_{a^{-1}a_i})w_j = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq 0}}^{l} \delta'_{i,j}(I_n \otimes A_{a_i})w_j.$$

Now for every $d = 1, \ldots, k$ we have

$$(I_n \otimes A_{a_d})w_{j_0} = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq 0}}^{l} \delta_{i,j,d}(I_n \otimes A_{a_i})w_j,$$

which implies that

$$\dim\ \mathrm{Span}\left\{(I_n \otimes A_{a_i})w_j \middle| i = 1, \ldots, k;\ j = 1, \ldots, l;\ j \neq j_0 \neq 1\right\} = s.$$

Now, by induction it can be proved that there exist $i_1 = 1, i_2, \ldots, i_{(s/k)}$ such that

$$\dim\ \mathrm{Span}\left\{(I_n \otimes A_i)w_{i_q} \middle| i = 1, \ldots, k; q = 1, \ldots, s/k\right\} = s.$$

Then, as in (11), we have

$$\mathrm{rowrank}(\{B(w_1), B(w_{i_2}), \ldots, B(w_{i_{(s/k)}})\}) = s. \qquad \square$$

Let $\mathbf{A}$ be an algebra, and let $\{a_1, \ldots, a_k\}$ be a basis for $\mathbf{A}$. For $v_1, \ldots, v_l \in F^{nk}$ we define

$$L_{\mathbf{A}}(\{v_1, \ldots, v_l\}) = \mathrm{Span}\left(\left\{(I_n \otimes A_{a_i})v_j \middle| i = 1, \ldots, k\ ;\ j = 1, \ldots, l\right\}\right).$$

The following theorem gives a lower bound for the complexity of a subset $B \subseteq \mathrm{Span}(\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}))$.

THEOREM 1. *Let* $\mathbf{A}$ *be a division algebra,* $V = \{v_1, \ldots, v_l\} \subseteq F^{n \dim \mathbf{A}}$ *and* $B = \{B(v_1), \ldots, B(v_l)\}$. *Then*

$$\mu(B) \geq \mu_0 = \dim\ \mathrm{Span}(V) + \left(1 - \frac{1}{\dim \mathbf{A}}\right) \dim\ L_{\mathbf{A}}(V).$$

(1) *If* $\mu(B) = \mu_0$, *then* $B \in DSCS_{SLA}$.
(2) *If* $\mu(B) = \mu_0 + 1$, *then* $B \in DSC_{SLA}$.
(3) *If* $\mu(B) > \mu_0 + 1$, *then for any set of matrices* C *we have*

$$\mu(B \oplus C) \geq \mu_0 + 1 + \mu(C).$$

Proof. We shall prove that $B \in DS^*$. By Lemma 15, for any nonsingular $l \times l$ matrix N and any $B(w_1) \in B[N]$, there exist $r - 1 = (\dim\ L_{\mathbf{A}}(\{v_1, \ldots, v_l\}) / \dim \mathbf{A}) - 1$ matrices $B(w_2), \ldots, B(w_r) \in B[N]$ such that

$$\mathrm{rowrank}(\{B(w_1), \ldots, B(w_r)\}) \geq \dim\ L_{\mathbf{A}}(\{v_1, \ldots, v_l\}).$$

Then by Lemma 5,

$$\mu(B) \geq \dim(B) + \left(1 - \frac{1}{\dim \mathbf{A}}\right) \dim L_{\mathbf{A}}(V)$$
$$= \dim \operatorname{Span}(V) + \left(1 - \frac{1}{\dim \mathbf{A}}\right) \dim L_{\mathbf{A}}(V).$$

By Lemma 6, the results (1) through (3) follow. □

THEOREM 2. *Let* $\mathbf{A}$ *be a direct sum of division algebras* $\mathbf{D}_1 \times \cdots \times \mathbf{D}_k$. *Then for any set of matrices* C,

$$\mu((\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A})) \oplus C) \geq (2 \dim \mathbf{A} - k)n + \mu(C).$$

If the bound is tight, then $\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) \in DSCS_{SLA}$.

Proof. By Theorem 1, we have for $B = \mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}_i)$ that $\dim B = n \dim \mathbf{D}_i$ and $\dim L_{\mathbf{D}_i}(V) = n \dim \mathbf{D}_i$, and therefore

$$\mu((\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{D}_i)) \oplus C) \geq (2 \dim(\mathbf{D}_i) - 1)n + \mu(C).$$

Since by Lemma 1 and (10), we have $\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) = \bigoplus_{i=1}^{w} (\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{D}_i))$, the result follows. □

For a matrix S we denote the linear space spanned by the columns and the rows of S by $L_{\text{col}}(S)$ and $L_{\text{row}}(S)$, respectively. We have:

LEMMA 16. *Let* $\mathbf{A}$ *be a division algebra of dimension* k. *Then for any nonsingular* $(nk \times nk)$ *matrix* N *and for any* $C_1 \in (\mathbf{I}_n^D \otimes \mathbf{C}(\mathbf{A}))[N]$, *there exist* $n-1$ *matrices* $C_2, \ldots, C_n \in (\mathbf{I}_n^D \otimes \mathbf{C}(\mathbf{A}))[N]$ *such that*

$$L_{\text{col}}[C_1 | \ldots | C_n] = F^{nk}.$$

And for any nonsingular $(nk \times nk)$ *matrix* N *and for any* $C_1 \in (\mathbf{I}_n^E \otimes \mathbf{C}(\mathbf{A}))[N]$, *there exist* $n-1$ *matrices* $C_2, \ldots, C_n \in (\mathbf{I}_n^E \otimes \mathbf{C}(\mathbf{A}))[N]$ *such that*

$$L_{\text{row}}[C_1 | \ldots | C_n] = F^{nk}.$$

Proof. Since $\mathbf{A}$ is a division algebra, $\mathbf{A}$ is a Frobenuis algebra, by Lemma 12. Therefore, by Lemmas 11 and 2 and since $\mathbf{B}(\mathbf{A}^-) = \mathbf{B}(\mathbf{A})^T$,

$$\mathbf{I}_n^D \otimes \mathbf{C}(\mathbf{A}) \equiv \mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}^-) \quad \text{and} \quad \mathbf{I}_n^E \otimes \mathbf{C}(\mathbf{A}) \equiv \mathbf{I}_n^E \otimes \mathbf{B}(\mathbf{A}^-) = (\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}))^T.$$

Therefore, by Lemma 15, the result follows. □

LEMMA 17. *Let* $\mathbf{A}$ *be a local algebra of dimension* k *and* $S = \mathbf{I}_n^D \otimes \mathbf{C}(\mathbf{A})$. *There exists a linear space* $S_1 \subseteq \operatorname{Span}(S)$ *such that for any nonsingular* $(nk \times nk)$ *matrix* N *and any* $C_1 \in S[N] \cap (\operatorname{Span}(S) \backslash S_1)$ *there exist* $C_2, \ldots, C_n \in S[N] \cap (\operatorname{Span}(S) \backslash S_1)$, *where*

$$\operatorname{colrank}\{C_1, \ldots, C_n\} = n \dim \mathbf{A}.$$

This lemma is also true for $S' = \mathbf{I}_n^E \otimes \mathbf{C}(\mathbf{A})$ with rowrank.

Proof. We shall use the basis in Lemma 13. Let $\mathbf{C}(\mathbf{A}) = \{A_{a_1}, \ldots, A_{a_k}, A_{a_{k+1}}, \ldots, A_{a_{rk}}\}$, $S_1 = \mathbf{I}_n^D \otimes \operatorname{Span}\{A_{a_{k+1}}, \ldots, A_{a_{rk}}\}$ and $S_2 = (\operatorname{Span}(S) \backslash S_1)$. By Lemma 13, taking the first k columns in A_{a_i}, $i = 1, \ldots, k$, we get $[\tilde{A}_i^T | 0_{k \times k} | \ldots | 0_{k \times k}]^T$, where $\mathbf{C}(\mathbf{A}/\text{rad } \mathbf{A}) = \{\tilde{A}_i\}$.

Taking the first k columns of A_{a_i}, $i > k$, we get $[0_{k\times k}|\ldots|0_{k\times k}]^T$. Since $\mathbf{A}$ is local, $\mathbf{A}/\text{rad}\,\mathbf{A}$ is a division algebra, and therefore we can apply Lemma 16 to it.

Let N be any nonsingular $(nk\times nk)$ matrix. Observing the first k columns of the matrices in $(\mathbf{I}_n^D\otimes\mathbf{C}(\mathbf{A}))[N]$, we conclude by Lemma 16 that for any $C_1\in(\mathbf{I}_n^D\otimes\mathbf{C}(\mathbf{A}))[N]\cap S_2$ there exist $C_2,\ldots,C_n\in(\mathbf{I}_n^D\otimes\mathbf{C}(\mathbf{A}))[N]\cap S_2$, such that $\dim\ L_{\text{col}}(\text{first } k \text{ columns in } C_1,\ldots,C_n)=nk$. In fact, it can be easy to verify that

$$\begin{aligned}&L_{\text{col}}(\text{first } k \text{ columns in } C_1,\ldots,C_n)\\&\quad=\left\{(v_1,0_{\dim\mathbf{A}-k},v_2,0_{\dim\mathbf{A}-k},\ldots,v_n,0_{\dim\mathbf{A}-k})^T|(v_1,v_2,\ldots,v_n)\in F^{nk}\right\},\end{aligned}$$

where $0_j=(0,\overset{j\text{times}}{\ldots},0)$. Since $[C_1|\ldots|C_n]$ is of the form

$$\begin{pmatrix}
C_{1,1} & & & * & & C_{1,n} & & & *\\
& C_{1,1} & & & \cdots & & C_{1,n} & & \\
& & \ddots & & \cdots & & & \ddots & \\
0 & & & C_{1,1} & & 0 & & & C_{1,n}\\
C_{2,1} & & & * & & C_{2,n} & & & *\\
& C_{2,1} & & & \cdots & & C_{2,n} & & \\
& & \ddots & & \cdots & & & \ddots & \\
0 & & & C_{2,1} & & 0 & & & C_{2,n}\\
& \vdots & & & \vdots & & \vdots & & \\
& \vdots & & & \vdots & & \vdots & & \\
& \vdots & & & \vdots & & \vdots & & \\
& \vdots & & & \vdots & & \vdots & & \\
C_{n,1} & & & * & & C_{n,n} & & & *\\
& C_{n,1} & & & \cdots & & C_{n,n} & & \\
& & \ddots & & \cdots & & & \ddots & \\
0 & & & C_{n,1} & & 0 & & & C_{n,n}
\end{pmatrix},$$

it follows that

$$L_{\text{col}}\begin{pmatrix}C_{1,1} & \cdots & C_{1,n}\\ \vdots & \vdots & \vdots\\ C_{n,1} & \cdots & C_{n,n}\end{pmatrix}=F^{nk}.$$

Therefore,

$$\begin{aligned}&L_{\text{col}}(i\text{th set of } k \text{ columns of } C_1,\ldots,C_n)\\&\quad=\{(*_{(i-1)k},v_1,0_{i(\dim\mathbf{A}-k)},*_{(i-1)k},v_2,0_{i(\dim\mathbf{A}-k)},\ldots,*_{(i-1)k},v_n,0_{i(\dim\mathbf{A}-k)})^T|\\&\qquad\qquad(v_1,v_2,\ldots,v_n)\in F^{nk}\},\end{aligned}$$

where $*_{(i-1)k}$ represents $(i-1)k$ entries with some possible nonzero elements from the field. Then obviously,

$$\begin{aligned}L_{\text{col}}(C_1,\ldots,C_n)&=L_{\text{col}}(\text{first set of } k \text{ columns of } C_1,\ldots,C_n)\oplus_F\\&\ L_{\text{col}}(\text{second set of } k \text{ columns of } C_1,\ldots,C_n)\oplus_F\\&\ \vdots\\&\ L_{\text{col}}(\text{last set of } k \text{ columns of } C_1,\ldots,C_n)=F^{n\,\dim\mathbf{A}}.\end{aligned}$$

This implies that $\text{colrank}\{C_1,\ldots,C_n\}=n\ \dim\ \mathbf{A}$. □

Lemma 17 implies the following theorem.

THEOREM 3. *Let* $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ *be an algebra where each* $\mathbf{A}_i$ *is local algebra. If* $\mathbf{A}$ *is a Frobenuis algebra, then for any set of matrices* C *we have*

$$\mu([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus C) \geq (2\dim\ \mathbf{A} - k)n + \mu(C). \tag{14}$$

If $\mathbf{A}$ *is an algebra of* n*-minimal complexity, then* (14) *holds for equality and*

$$[x_1 y, \ldots, x_n y]_{\mathbf{A}} \in DSCS_{SLA}. \tag{15}$$

Proof. We have, by Lemmas 1 and 10, that

$$[x_1 y, \ldots, x_n y] = \mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) = \bigoplus_{i=1}^{k} (\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}_i)).$$

Then by Lemmas 4 and 12 it is sufficient to prove the theorem for each of the local Frobenuis algebra $\mathbf{A}_i$.

Now since $\mathbf{A}_i$ is a Frobenuis algebra, by Lemma 14, $\mu(\mathbf{I}_n \otimes \mathbf{B}(\mathbf{A}_i)) = \mu(\mathbf{I}_n \otimes \mathbf{C}(\mathbf{A}_i))$. By Lemmas 5, 6, and 17 the result follows. □

THEOREM 4. *Let* $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ *be an algebra where each* $\mathbf{A}_i$ *is a local algebra. Then for any set of matrices* C *we have*

$$\delta([x_1 y, \ldots, x_n y]_{\mathbf{A}} \oplus C) \geq (2\dim\ \mathbf{A} - k)n + \delta(C). \tag{16}$$

If $\mathbf{A}$ *is an algebra of* n*-minimal rank, then* (16) *hold for equality and*

$$[x_1 y, \ldots, x_n y]_{\mathbf{A}} \in DSCS_{BA}.$$

Proof. As in the proof of Theorem 3, $[x_1 y, \ldots, x_n y] = \mathbf{I}_n^E \otimes \mathbf{B}(\mathbf{A})$ and

$$\delta((\mathbf{I}_n^E \otimes \mathbf{C}(\mathbf{A}^-)) \oplus C^{TD}) \geq (2\dim\ \mathbf{A} - k) + \delta(C^{TD}).$$

Then, by Lemma 4, and since $\mathbf{I}_n^{ED} = \mathbf{I}_n^E$, we have

$$\delta((\mathbf{I}_n^E \otimes \mathbf{B}(\mathbf{A}^-)) \oplus C^T) \geq (2\dim\ \mathbf{A} - k) + \delta(C^T).$$

And by Lemma 14 and since $\mathbf{I}_n^{ET} = \mathbf{I}_n^D$ and $\mathbf{B}(\mathbf{A}^-)^T = \mathbf{B}(\mathbf{A})$, we have

$$\delta((\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A})) \oplus C) \geq (2\dim\ \mathbf{A} - k) + \delta(C).$$

If $\mathbf{A}$ is of n-minimal rank, then $\mathbf{I}_n^D \otimes \mathbf{C}(\mathbf{A}) \in DSCS_{BA}$, and therefore, by Lemma 7, $\mathbf{I}_n^D \otimes \mathbf{B}(\mathbf{A}) \in DSCS_{BA}$. □

4. Complexity of one bilinear form over algebras. In this section we shall study the complexity of one bilinear form over algebras.

Let H be an $n \times m$ matrix, and let $\mathbf{A}$ be an algebra. Then:

LEMMA 18. $\mu(\{H\} \otimes \mathbf{B}(\mathbf{A})) = \mu(\mathbf{I}_r \otimes \mathbf{B}(\mathbf{A}))$, *where* $r = \mathrm{rank}(H)$.

Proof. Let N and M be nonsingular matrices such that $NHM = \mathrm{diag}(I_r, 0)$. Then

$$(N \otimes I_k)(\{H\} \otimes \mathbf{B}(\mathbf{A}))(M \otimes I_k) = \mathrm{diag}(\mathbf{I}_r, \{0\}) \otimes \mathbf{B}(\mathbf{A}),$$

which implies the result. □

The main results of this section are:

COROLLARY 1. *Let $\mathbf{A} = \mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ be an algebra where each $\mathbf{A}_i$ is a local algebra. Let H be an $n \times m$ matrix. Then for every set of matrices C we have*

$$\delta([x^T Hy]_{\mathbf{A}} \oplus C) \geq (2 \dim\ \mathbf{A} - k)\mathrm{rank}(H) + \delta(C).$$

If $\mathbf{A}$ is an algebra of rank(H)*-minimal rank, then*

$$[x^T Hy]_{\mathbf{A}} \in DSCS_{BA}.$$

Proof. Let $r = \mathrm{rank}(H)$. In Lemma 17 we proved this theorem for $\mathbf{I}_r^D \otimes \mathbf{C}(\mathbf{A})$. Then by Lemmas 1 and 4 it is also true for

$$(\mathbf{I}_r^D \otimes \mathbf{C}(\mathbf{A}))^D = \mathbf{I}_r \otimes \mathbf{B}(\mathbf{A}) = [x_1 y_1 + \cdots + x_n y_n]_{\mathbf{A}} \equiv [x^T Hy]_{\mathbf{A}}. \qquad \square$$

THEOREM 5. *Let $\mathbf{A}$ be a commutative algebra. Let H be $n \times m$ matrix. Then*

$$\mu([x^T Hy]_{\mathbf{A}}) \geq (2 \dim\ \mathbf{A} - I(\mathbf{A}))\mathrm{rank}(H).$$

If $\mathbf{A}_1$ and $\mathbf{A}_2$ are commutative algebras of minimal rank, then

$$([x^T H_1 y]_{\mathbf{A}_1}, [x^T H_2 y]_{\mathbf{A}_2}) \in DSCS_{SLA}.$$

Proof. Let $r = \mathrm{rank}(H)$. By Lemmas 4 and 18, we have

$$\mu([x^T Hy]_{\mathbf{A}}) = \mu(\mathbf{I}_r \otimes \mathbf{B}(\mathbf{A})) \geq \frac{1}{2}\delta(\mathrm{diag}((\mathbf{I}_r \otimes \mathbf{B}(\mathbf{A})), (\mathbf{I}_r \otimes \mathbf{B}(\mathbf{A}))^T)).$$

Since $\mathbf{A}$ is commutative, we have $\mathbf{B}(\mathbf{A})^T = \mathbf{B}(\mathbf{A})$, and by Theorem 5, we have

$$\mu([x^T Hy]_{\mathbf{A}}) \geq \frac{1}{2}\delta(\mathbf{I}_{2r} \otimes \mathbf{B}(\mathbf{A})) = (2 \dim\ \mathbf{A} - k)\mathrm{rank}(H).$$

Now by Lemma 8, $([x^T H_1 y]_{\mathbf{A}_1}, [x^T H_2 y]_{\mathbf{A}_2}) \in DSCS_{SLA}$. $\square$

REFERENCES

[1] A. A. ALBERT, *Structure of algebras*, Amer. Math. Soc. Colloq. Publ. 24, 1939.

[2] A. AVERBUCH, Z. GALIL AND S. WINOGRAD, *Classification of all the minimal bilinear algorithm for computing the coefficient of the product of two polynomials modulo a polynomial in the algebra $G[u]/\langle u^n \rangle$*, Theoret. Comput. Sci., 86 (1991), pp. 143–203.

[3] ———, *Classification of all the minimal bilinear algorithm for computing the coefficient of the product of two polynomials modulo a polynomial in the algebra $G[u]/\langle Q(u)^l \rangle$, $l > 1$*, Theoret. Comput. Sci., 58 (1988), pp. 17–56.

[4] A. ALDER AND V. STRASSEN, *On the algorithmic complexity of associative algebras*, Theoret. Comput. Sci., 15 (1981), pp. 201–211.

[5] L. AUSLANDER AND S. WINOGRAD, *The multiplicative complexity of certain semilinear systems defined by polynomials*, Adv. Appl. Math., 1 (1980), pp. 157–299.

[6] N. H. BSHOUTY, *On the extended direct sum conjecture*, Proceedings 21st Annual ACM Symposium on Theory of Computing, May 1989.

[7] ———, *On the regular representation of algebras*, University of Calgary.

[8] ———, *On the direct sum conjecture in the straight line model*, ESA '93, October 1993.

[9] C. M. FEDUCCIA AND Y. ZALCSTEIN, *Algebras having linear multiplicative complexity*, J. Assoc. Comput. Mach., 24 (1977), pp. 311–331.

[10] H. F. GROOTE, *Characterization of division algebras of minimal rank and the structure of their algorithm varieties*, SIAM J. Comput., 12 (1983), pp. 101–117.

[11] H. F. GROOTE, *Lectures on the complexity of bilinear problems*, Lecture Notes in Comput. Sci. 245, Springer, Berlin, 1987.

[12] H. F. GROOTE AND J. HEINTZ, *Commutative algebra of minimal rank*, Linear Algebra Appl., 55 (1983), pp. 37–68.

[13] W. HARTMANN, *On the multiplicative complexity of modules over associative algebras*, SIAM J. Appl. Math., 14 (1985), pp. 383–395.

[14] J. HOPCROFT AND J. MUNSINSKI, *Duality applied to the complexity of matrix multiplication*, SIAM J. Comput., 2 (1973), pp. 159–173.

[15] J. HEINTZ AND J. MORGENSTERN, *On associative algebras of minimal rank*, Proc. of the AAECC-2 Conference, Grenoble 1988.

[16] J. JAJA, *On the complexity of bilinear forms with commutativity*, SIAM. J. Comput., 9, 4, (1980), pp. 713–728.

[17] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math. 264, (1973), pp. 184–202.

[18] S. WINOGRAD, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.

[19] C. W. CURTIS AND I. RIENER, *Representation theory of finite groups and associative algebras*, John Wiley, New York, 1988.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 834–851, August 1994

009

A NEW LOWER BOUND TECHNIQUE AND ITS APPLICATION: TIGHT LOWER BOUND FOR A POLYGON TRIANGULATION PROBLEM*

PRAKASH RAMANAN†

Abstract. A new technique for obtaining lower bounds on the worst-case time-complexity of optimization problems in the linear decision tree model of computation is presented. This technique is then used to obtain a tight $\Omega(n \log n)$ lower bound for a problem of finding a minimum cost triangulation of a convex polygon with weighted vertices. This problem is similar to the problem of finding an optimal order of computing a matrix chain product. If the lower bound technique could be extended to bounded degree algebraic decision trees, a tight $\Omega(n \log n)$ lower bound for this latter problem would be obtained.

Key words. optimization problem, time-complexity, algebraic decision tree model, lower bound, polygon triangulation, matrix chain product

AMS subject classifications. 68Q25, 68Q20, 68Q05

1. Introduction.

Let P be an optimization problem whose input instances are points $\vec{x} = (x_1, x_2, \ldots, x_n)$ in (the positive orthant of) the n-dimensional real space R^n, for some $n \geq 1$. Let $S(P)$ be the finite set of solutions of P. Associated with each $s \in S(P)$ is a polynomial cost function $c_s(\vec{x})$. P can be stated as follows:

Given $\vec{x} \in R^n$, find $s \in S(P)$ such that $c_s(\vec{x})$ is as small as possible.

Suppose that each $s \in S(P)$ is the unique optimal solution for some instance $\vec{x} \in R^n$. Then $|S(P)|$ is a lower bound on the number of leaves of any decision tree that solves P. The following is well known.

THEOREM 1. *Any decision tree that solves P has $\Omega(\log |S(P)|)$ height.*

For many problems, the lower bound obtained using the above theorem is unsatisfactory. In §2, we present a new technique for obtaining a lower bound on the worst-case time-complexity of P, in the linear decision tree model of computation. In §3, we use this technique to obtain a tight $\Omega(n \log n)$ lower bound for a problem of finding a minimum cost triangulation of a convex polygon with weighted vertices. This problem is similar to the problem of finding an optimal order of computing a matrix chain product. If we could extend our lower-bound technique to bounded degree algebraic decision trees, we would have a tight $\Omega(n \log n)$ lower bound for this latter problem. In §4, we present our conclusions.

2. The lower-bound technique.

The model of computation we refer to is the *algebraic decision tree model* (see [1], [2]). An input instance is a sequence $\vec{x} = (x_1, x_2, \ldots, x_n) \in R^n$. An algebraic decision tree is a ternary tree. Each internal node v of the decision tree contains a comparison of the form $f_v(\vec{x}) : 0$, for some nonzero polynomial function f_v of n arguments; the three branches emanating from v are labeled with $<$, $=$, and $>$, corresponding to the three possible outcomes of the comparison. Each leaf l of the decision tree contains a solution sol(l). For any input $\vec{x}$, the computation moves from the root down the decision tree, at each internal node v testing and branching according to whether $f_v(\vec{x})$ is < 0, $= 0$, or > 0, until a leaf l is reached; then sol(l) is the output of the decision tree. The worst-case time-complexity of the decision tree is its height. An algebraic decision tree is said to be of *degree d* if the degrees of the polynomials f_v are all at most d. An algebraic decision tree is said to be of

*Received by the editors October 29, 1990; accepted for publication (in revised form) May 15, 1993. This research was supported in part by National Science Foundation grant CCR 88-03241, Lawrence Livermore National Laboratory's Institute for Scientific Computing Research grant ISCR 89-28, and by a Summer Research Fellowship from the Liberal Arts and Sciences College of the Wichita State University.

†Department of Computer Science, Wichita State University, Wichita, KS 67260-0083 (ramanan@cs.twsu.edu).

bounded degree if it is of degree d for some constant d independent of n. The *linear decision tree model* is a special case, where each of the polynomials is restricted to be of degree one.

Let P be an optimization problem. For an instance $\vec{x} \in R^n$, let $\mathrm{sol}(\vec{x}) \subseteq S(P)$ be the set of all optimal solutions for $\vec{x}$. For $s \in S(P)$, let $\mathrm{set}(s)$ be the set of all instances $\vec{x}$ such that $s \in \mathrm{sol}(\vec{x})$. Let D be a decision tree that solves P. When presented with an instance $\vec{x}$, D needs to output some $s \in \mathrm{sol}(\vec{x})$.

Suppose that for each $s \in S(P)$, the cost function $c_s(\vec{x})$ is a linear function of $\vec{x}$; then P can be solved by a linear decision tree. There exists a partition of R^n defined by hyperplanes, into convex polyhedral regions, such that each $s \in S(P)$ is associated with a unique polyhedron; so $\mathrm{set}(s)$ is a closed convex polyhedron. The solution associated with a polyhedron is uniquely optimal for each instance in the interior of that polyhedron. For an instance $\vec{x}$ on the boundary of two or more polyhedra, $\mathrm{sol}(\vec{x})$ consists of the solutions associated with each of the polyhedra which share that boundary. Consider the following problem of testing the optimality:

Given $s \in S(P)$ and $\vec{x} \in R^n$, determine if s is optimal for instance $\vec{x}$.

This is the problem of testing if $\vec{x}$ is inside (or on the boundary of) the polyhedron $\mathrm{set}(s)$.

Let $K = \{1, 2, \ldots, k\}$ for some positive integer k, and let $C \subseteq R^n$ be the convex polyhedron defined by the set of linear inequalities $\{p_i(\vec{x}) \leq 0 | i \in K\}$. For $F \subseteq K$, let $C_F = \{\vec{x} \in R^n | p_i(\vec{x}) = 0 \text{ for } i \in F \text{ and } p_i(\vec{x}) < 0 \text{ for } i \in K - F\}$. If C_F is nonempty, then it is called a *face* of C (the face defined by F). The *dimension* of a face is the dimension of the smallest affine space that contains the face (see [6]); note that if the dimension is at least one, then the face is an open subset of this affine space. A zero-dimensional face is a vertex, a one-dimensional face is an edge, and an $(n-1)$-dimensional face is called a *facet*. According to our definition, a face cannot be contained in another (higher dimensional) face. Yao and Rivest [7] proved the following.

THEOREM 2 [7]. *Let C be a fixed convex polyhedron in R^n. Any linear decision tree that gives $\vec{x} \in R^n$, determines if $\vec{x} \in C$ has $\Omega(\log f)$ height, where f is the total number of faces (of all dimensions) of C.*

We say that $S' \subseteq S(P)$ is a *realizable optimal solution set* (ross), if there exists an instance $\vec{x}$ such that $S' = \mathrm{sol}(\vec{x})$; that is, if there exists an instance of P for which each $s \in S'$ is an optimal solution and no $s \in (S(P) - S')$ is optimal. Each ross corresponds to a face in the polyhedral subdivision of R^n corresponding to P. Let ROSS(P) be the set of all ross's of P. We have the following.

THEOREM 3. *Any linear decision tree that solves P has $\Omega(\log |\mathrm{ROSS}(P)|)$ height.*

We can prove this theorem using Theorem 2; we give a direct proof. We need the following definitions and notations. Let D be a linear decision tree that solves P. For a leaf l in D, let ξ_l be the path from the root to l, and let $\mathrm{set}(l)$ be the set of all instances that follow this path. For any $s \in S(P)$, let $L(s)$ be the set of all leaves l in D such that $\mathrm{sol}(l) = s$. Let

$$L'(s) = \{l \in L(s) | \xi_l \text{ does not contain any branch labeled } =\}.$$

Let $L(D)$ be the set of all leaves in D, and let

$$L'(D) = \{l \in L(D) | \xi_l \text{ does not contain any branch labeled } =\}.$$

For a set $X \subseteq R^n$, let *closure*(X) denote the smallest closed set in R^n containing X; that is, *closure*$(X) = X \cup$ *boundary*(X).

Proof of Theorem 3. There is a natural one-to-one correspondence between $\mathrm{ROSS}(P)$ and the set of faces in the polyhedral subdivision of R^n corresponding to P; so, $|\mathrm{ROSS}(P)|$ is the total number of faces in the subdivision. Let D be a linear decision tree of height h that solves P. We will prove that

$$(0)\qquad |\mathrm{ROSS}(P)| \leq \sum_{l\in L'(D)} \text{number of faces of } closure(set(l)) \leq 2^{2h}.$$

This will imply the desired result.

Consider a leaf $l \in L'(s)$ for some $s \in S(P)$. Clearly, set(l) and set(s) are open and closed convex polyhedra, respectively, and set(l) $\subseteq$ set(s); so, no face of set(s) can intersect with a higher dimensional face of *closure*(set(l)). Because set(s) $= \cup_{l\in L'(s)}$ *closure*(set(l)), each face of set(s) must contain a face (of the same dimension) of *closure*(set(l)) for some $l \in L'(s)$. So number of faces of set(s) $\leq \sum_{l\in L'(s)}$ number of faces of *closure*(set(l)). Hence, $\sum_{s\in S(P)}$ number of faces of set(s) $\leq \sum_{l\in L'(D)}$ number of faces of *closure*(set(l)). Also, $|\mathrm{ROSS}(P)| \leq \sum_{s\in S(P)}$ number of faces of set(s). The latter two inequalities, together, imply the first of the two inequalities in (0).

At each node v on the path ξ_l, let the comparison performed be of the form $f_v(\vec{x}) : 0$. A face of *closure*(set(l)) corresponds to some of these comparisons resulting in equality, and the others resulting in strict inequality. Hence, *closure*(set(l)) has at most 2^h faces. This, together with $|L'(D)| \leq 2^h$, implies the second inequality in (0). □

Note that, with slight modifications, the above lower bound proof also holds for problem P restricted to instances $\vec{x} \in R^n - A$, where A is any set of measure zero. In particular, it holds for problem P restricted to instances $\vec{x}$ such that $|\mathrm{sol}(\vec{x})| = 1$. With slight modifications, it even holds for the following problem of testing if the optimal solution is unique: Given $\vec{x} \in R^n$, determine if $|\mathrm{sol}(\vec{x})| = 1$.

When $c_s(\vec{x})$ is nonlinear in $\vec{x}$ for some s, the subdivision of R^n corresponding to P need not be a polyhedral subdivision. In this case, P cannot be solved by any linear decision tree, and any lower bound based on Theorem 3 would be useless. It would be nice if the result of Theorem 3 could be extended to bounded degree algebraic decision trees.

Now, we will extend the result of Theorem 3 (as well as any similar future results for algebraic decision trees) to the case when the input is restricted to contain only integers. $\vec{x} \in R^n$ is said to be *rational* if each component of $\vec{x}$ is rational. $\vec{x}$ is said to be *integral* if each component of $\vec{x}$ is integral. Let P_I be the integer version of problem P, that is, problem P restricted to integral instances $\vec{x} \in I^n$, where I^n is the set of all n tuples of (positive) integers. An algebraic decision tree that solves P_I could output a wrong solution when given an input $\vec{x} \in R^n - I^n$. We now prove that for many problems P, the lower bound result of Theorem 3 also holds for P_I.

Let sign(u) be 1, 0, or -1 depending on whether the real number u is positive, zero, or negative, respectively. A polynomial $f(\vec{x})$ is said to be *scale invariant* if $\mathrm{sign}(f(\vec{x})) = \mathrm{sign}(f(\alpha\vec{x}))$ for all $\vec{x} \in R^n$ and real $\alpha > 0$; if all the monomials in $f(\vec{x})$ have the same degree, then clearly $f(\vec{x})$ is scale invariant. A set $X \subseteq R^n$ is said to be *scale invariant* if $\vec{x} \in X \rightarrow \alpha\vec{x} \in X$, for all real $\alpha > 0$. An optimization problem P is said to be *scale invariant* if set(s) is scale invariant for all $s \in S(P)$; if $c_s(\vec{x}) - c_{s'}(\vec{x})$ is scale invariant for all $s, s' \in S(P)$, then clearly P is scale invariant.

We need the following definitions from Yao [8]. For a polynomial $f(\vec{x})$, let $\phi f(\vec{x}) = \lim_{\lambda\to 0} \lambda^d f(\vec{x}/\lambda)$, where d is the degree of f; $\phi f(\vec{x})$ is the sum of the degree d monomials in $f(\vec{x})$, and so it is scale invariant. For an algebraic decision tree D, let $\phi(D)$ be the decision tree obtained from D by replacing at each internal node v, the comparison $f_v(\vec{x}) : 0$ by the comparison $\phi f_v(\vec{x}) : 0$. The node in $\phi(D)$ corresponding to $v \in D$ will be referred to as $\phi(v)$. Using arguments similar to those of Yao [8, Lemma 3], we prove the following.

LEMMA 1. *Let P be an optimization problem that is scale invariant. Let D be an algebraic decision tree that solves P_I. For $\vec{x} \in R^n$, if $\vec{x} \in$ set($\phi(l)$) for some $l \in L'(D)$, then* sol($\phi(l)$) $\in$ sol($\vec{x}$); *so there exists a set $A \subseteq R^n$ of measure zero, such that $\phi(D)$ correctly solves P for any $\vec{x} \in R^n - A$.*

Proof. Let $\vec{x} \in \text{set}(\phi(l))$ for some $l \in L'(D)$. Each connected component of $\text{set}(\phi(l))$ is an open set of dimension n. So, we can pick an $\varepsilon > 0$ such that

$$B_\varepsilon = \{\vec{y} \in R^n \mid \|\vec{y} - \vec{x}\| < \varepsilon\} \subseteq \text{set}(\phi(l)).$$

Let $\vec{z} \in B_\varepsilon$; then $\vec{z} \in \text{set}(\phi(l))$. Because $\phi(l) \in L'(\phi(D))$, we have

$$\text{sign}(\phi f_v(\vec{z})) = \text{sign}(\phi f_v(\vec{x})) \neq 0$$

for all $v \in \xi_l$. We pick $\vec{z}$ to be rational. Let N be a positive integer such that $N\vec{z}$ is integral. Because $\phi f(\vec{x})$ is scale invariant, we have

$$\text{sign}(\phi f_v(N\vec{z})) = \text{sign}(\phi f_v(\vec{z})) \neq 0$$

for all $v \in \xi_l$. We will pick N large enough such that

$$\text{sign}(\phi f_v(N\vec{z})) = \text{sign}(f_v(N\vec{z}))$$

for all $v \in \xi_l$. Putting together all the preceding equalities, we have that

$$\text{sign}(f_v(N\vec{z})) = \text{sign}(\phi f_v(\vec{x}))$$

for all $v \in \xi_l$; so $N\vec{z} \in \text{set}(l)$. Because D correctly solves P_I, $\text{sol}(l) \in \text{sol}(N\vec{z})$. Because P is scale invariant, $\text{sol}(l) \in \text{sol}(\vec{z})$; this is true for any rational $\vec{z} \in B_\varepsilon$.

An optimization problem corresponds to a subdivision of R^n into regions such that each solution is optimal over some region; the boundaries between the regions are defined by polynomial equations. Because $\text{sol}(\phi(l)) = \text{sol}(l) \in \text{sol}(\vec{z})$ for any rational $\vec{z} \in B_\varepsilon$, this implies that $\text{sol}(\phi(l)) \in \text{sol}(\vec{x})$.

Finally, let $A = \cup_{l \in L(D)-L'(D)} \text{set}(\phi(l))$; $A \subseteq \{\vec{y} \in R^n \mid \phi f_v(\vec{y}) = 0 \text{ for some } v \in D\}$, and so A is of measure zero. $\phi(D)$ correctly solves P for any $\vec{x} \in R^n - A$. □

As a corollary, we have the following.

THEOREM 4. *Let P be an optimization problem that is scale invariant. Any linear decision tree that solves P_I has $\Omega(\log|\text{ROSS}(P)|)$ height.*

Proof. The proof follows from Lemma 1, Theorem 3, and the comment immediately following the proof of Theorem 3. □

Previously, there has been no natural problem for which a nontrivial lower bound has been obtained using Theorems 2–4. In this paper, we present such a lower bound. As a simple application of Theorems 3 and 4, consider the following problem:

(MIN) Given $\vec{x} = (x_1, x_2, \ldots, x_n) \in R^n$, find i, $1 \le i \le n$, such that $x_i = \min\limits_{1 \le j \le n} x_j$.

We have $S(\text{MIN}) = \{1, 2, \ldots, n\}$. It is well known that $n - 1$ comparisons are necessary and sufficient to solve (MIN), but the lower bound is based on ad hoc arguments. Theorem 1 yields only an $\Omega(\log n)$ lower bound. Note that any nonempty subset $S' \subseteq S(\text{MIN})$ is a ross. Hence, an $\Omega(n)$ lower bound follows from Theorem 3. A similar lower bound can be obtained using Theorem 2, but it is more tedious. Also, MIN is scale invariant. Hence, by Theorem 4, MIN_I also has an $\Omega(n)$ lower bound.

3. Tight lower bound for a polygon triangulation problem.

A convex polygon is specified by a list (i.e., a sequence) of vertices, in cyclic order, around the boundary of the polygon. Let $SP = (v_1, v_2, \ldots, v_n)$ be a convex polygon. An *edge* $v_i v_{i \bmod n+1}$, $1 \le i \le n$, is a straight line segment that connects the two adjacent vertices v_i and $v_{i \bmod n+1}$ of the polygon.

An arc $v_i v_j$ is a straight line segment that connects the two nonadjacent vertices v_i and v_j of the polygon. Each vertex v_i, $1 \leq i \leq n$, has a *weight* $w_i > 0$ associated with it. A *triangulation* T (of the interior) of SP consists of $n-2$ triangles formed by the n edges and $n-3$ nonintersecting arcs. The *cost* of a triangle $v_i v_j v_k$ is $c(w_i, w_j, w_k)$. The *cost of* T, denoted by $c_T(\vec{w})$, is the sum of costs of all the triangles in T. The triangulation problem we are interested in is as follows.

(TP)
Given $\vec{w} = (w_1, w_2, \ldots, w_n)$, find a triangulation T such that $c_T(\vec{w})$ is as small as possible.

There are C_{n-2} triangulations of SP, where C_k is the kth Catalan number, $C_k = 2^{\theta(k)}$ (see [3]). Hence, Theorem 1 yields only a trivial lower bound.

We consider two different cost functions. In the sum case, $c(w_i, w_j, w_k) = w_i + w_j + w_k$; in the *product case*, $c(w_i, w_j, w_k) = w_i w_j w_k$. These two versions of *TP* will be referred to as *TPS* and *TPP*, respectively. Note that for *TPS*, for any two triangulations T_1 and T_2, $c_{T_1}(\vec{w}) - c_{T_2}(\vec{w})$ remain unchanged if the cost function is redefined as follows: The *cost* of an arc $v_i v_j$ is $w_i + w_j$; $c_T(\vec{w})$ is the sum of costs of all the arcs in T. Also note that *TPS* can be solved by a linear decision tree. *TPP* cannot be solved by any degree two decision tree; this can be shown as follows: Let $n = 4$ and $w_1 < w_2 < w_4 < w_3$; to find an optimal triangulation of SP, we need to determine which of the two costs $w_1 w_3 (w_2 + w_4)$ and $w_2 w_4 (w_1 + w_3)$ is smaller. This cannot be determined by any degree two decision tree.

Hu and Shing [4], [5] considered the following matrix chain product problem.

(MCPP) Given integers $(w_1, w_2, \ldots, w_n)$, find an optimal order of computing the matrix chain product $M_1 \times M_2 \times \cdots \times M_{n-1}$, where M_i is of dimensions $w_i \times w_{i+1}$ and the cost of computing the product of a $w_i \times w_j$ matrix with a $w_j \times w_k$ matrix is $w_i w_j w_k$.

They showed that this problem is equivalent to TPP_I, and they presented an $O(n \log n)$ algorithm for *TPP*. This algorithm can be easily modified to solve *TPS*. In §3.1, we present a simpler exposition of their algorithm for *TPS* and *TPP*. In §3.2, based on this algorithm, we identify $n^{\theta(n)}$ ross's for *TPS* and *TPP*. Note that *TPS* and *TPP* are scale invariant. So an $\Omega(n \log n)$ lower bound for *TPS* and TPS_I in the linear decision tree model follows from Theorems 3 and 4, respectively. Note that any lower bound for *TPP* based on Theorem 3 would be useless, because *TPP* cannot be solved by any linear decision tree. If we could extend the result of Theorem 3 to bounded-degree algebraic decision trees, we would have a tight $\Omega(n \log n)$ lower bound for *TPP* and for $TPP_I \equiv MCPP$. In §3.3, we discuss the similarity and the relationship between *TPS* and *TPP*.

3.1. The algorithm. Let $SP = (v_1, v_2, \ldots, v_n)$ be a convex polygon. Without loss of generality, let v_1 be a vertex of smallest weight in SP; that is, $w_1 = \min_{1 \leq i \leq n} w_i$; such a vertex is said to be a *global minimum*. A *global maximum* vertex is defined analogously. A vertex v_i is said to be a *local minimum* if $w_i \leq \min(w_{i-1}, w_{i+1})$. A *local maximum* vertex is defined analogously. For the sake of simplicity, we make the following assumptions concerning SP:

(A1) v_1 is the unique vertex of smallest weight.

(A2) For $i < j$ such that $w_i = w_j$, there exists k, $i < k < j$, such that $w_k < w_i$.

Then the number of local minimum vertices and local maximum vertices are the same. SP is said to be *m-modal* if it has m local minimum vertices.

We let $w(v)$ denote the weight of a vertex v. An arc $r = v_i v_j$, $1 < i < j \leq n$, is called a *primary horizontal arc* (*ph-arc*) if $w_k \geq \max(w_i, w_j)$ for all k, $i < k < j$ (Hu and Shing [4] called it a *potential horizontal arc*). If r is a ph-arc, $SP(r) = (v_i, v_{i+1}, \ldots, v_j)$ is called the *upper subpolygon* of SP that is *bounded below* by r.

An arc $r = v_1v_j, 2 < j < n$, is called a *secondary horizontal arc* (*sh-arc*) if either of the following two conditions hold:

(a) $w_k \geq w_j$ for all k, $1 < k < j$. In this case, $SP(r) = (v_1, v_2, \ldots, v_j)$ is called the *upper subpolygon* of SP that is *bounded below* by r.

(b) $w_k \geq w_j$ for all $k > j$. In this case, $SP(r) = (v_1, v_j, v_{j+1}, \ldots, v_n)$ is called the *upper subpolygon* of SP that is *bounded below* by r.

An arc is called a *horizontal arc* (*h-arc*) if it is either a ph-arc or a sh-arc. Hu and Shing [4] showed that, under assumptions (A1) and (A2), there are exactly $n - 3$ h-arcs of SP, and no two of them intersect; so they constitute a triangulation of SP. Hu and Shing also presented a linear-time *One-Sweep algorithm* for finding all the h-arcs.

For two h-arcs r and r' of SP, r is said to be *below* r', denoted by $r < r'$ (equivalently, r' is *above* r, denoted by $r' > r$), if all the vertices of $SP(r')$ are also vertices of $SP(r)$. Two h-arcs are said to be *comparable* if one of them is below the other; otherwise they are said to be *incomparable*.

A *subpolygon* of SP is a polygon each of whose edges is either an edge or an h-arc of SP. Note that the h-arcs of a subpolygon of SP are those h-arcs of SP that are in the subpolygon.

If r_i, $1 \leq i \leq k$, are pairwise incomparable h-arcs, we let $SP(; r_1, r_2, \ldots, r_k)$ denote the subpolygon obtained from SP as follows. For each i, $1 \leq i \leq k$, remove from SP all the vertices of $SP(r_i)$ except the endpoints of r_i. This subpolygon is said to be *bounded above* by r_i, $1 \leq i \leq k$. If r is another h-arc such that $r < r_i$, $1 \leq i \leq k$, we let $SP(r; r_1, r_2, \ldots, r_k)$ denote the subpolygon obtained from $SP(r)$ in a similar manner, that is, $SP(r; r_1, r_2, \ldots, r_k) = (SP(r))(; r_1, r_2, \ldots, r_k)$. This subpolygon is said to be *bounded below* by r and *bounded above* by r_i, $1 \leq i \leq k$.

A *fan* of a (sub)polygon is a triangulation of the (sub)polygon in which a vertex v of smallest weight is connected to all the other vertices; v is called the *center* of the fan. The following lemma and its corollary are from Hu and Shing [5, Lemma 1]. For the sake of completeness, we present their proof here.

LEMMA 2 [5]. *If an optimal triangulation of a polygon contains none of the ph-arcs, then it must be a fan of the polygon.*

Proof. The proof is by induction on the number of vertices in the polygon. For a polygon with three vertices, a triangulation contains no arcs, and so the statement of the lemma is vacuously true. For the basis step, consider the polygon (v_1, v_2, v_3, v_4) with four vertices, where v_1 is a global minimum. There are only two different triangulations, each containing one arc. The triangulation containing the arc v_1v_3 is a fan. So, let the triangulation containing the arc v_2v_4 be optimal; we will prove that v_2v_4 is a ph-arc. For *TPS*, we have $w_2 + w_4 \leq w_1 + w_3$. For *TPP*, we have $w_2w_4(w_1 + w_3) \leq w_1w_3(w_2 + w_4)$; dividing both sides of this inequality by $w_1w_2w_3w_4$, we get $1/w_1 + 1/w_3 \leq 1/w_2 + 1/w_4$. For both *TPS* and *TPP*, because v_1 is a global minimum, the corresponding inequality implies that v_3 is a global maximum. So v_2v_4 is a ph-arc.

For the induction hypothesis, let the statement of the lemma be true for all polygons with fewer that n vertices. Let $SP = (v_1, v_2, \ldots, v_n)$ be a polygon with n vertices, where v_1 is a global minimum. Let T be an optimal triangulation of SP that contains no ph-arcs. If T contains an arc v_1v_j for some $j, 2 < j < n$, then consider the polygons $SP_1 = (v_1, v_2, \ldots, v_j)$ and $SP_2 = (v_1, v_j, v_{j+1}, \ldots, v_n)$; let T_1 and T_2 be their respective optimal triangulations induced by T. Because v_1 is a global minimum in SP, any ph-arc of SP_1 or SP_2 is also a ph-arc of SP. Then, because T contains no ph-arcs of SP, T_1 and T_2 contain no ph-arcs. Hence, by induction hypothesis, T_1 and T_2 must be fans of SP_1 and SP_2, respectively, with center v_1. So, T must be a fan of SP with center v_1.

So, assume that T does not contain the arc v_1v_j for any j, $2 < j < n$; we will show that this leads to a contradiction. T must contain the arc v_2v_n; without loss of generality, let

$w_2 \leq w_n$. Let $SP' = (v_2, v_3, \ldots, v_n)$, and let T' be the optimal triangulation of SP' induced by T. We will consider two cases.

Case 1. v_2 is a global minimum vertex in SP'. Then, any ph-arc of SP' is also a ph-arc of SP. Because T contains no ph-arcs of SP, by induction hypothesis, T' must be a fan of SP' with center v_2. Among the vertices v_i, $3 \leq i < n$, let v_a be a vertex of smallest weight (see Fig. 1a). Because v_2v_n is not a ph-arc of SP, we have $w_a < w_n$. Let $SP'' = (v_1, v_2, v_a, v_{a+1}, \ldots, v_n)$, and let T'' be the optimal triangulation of SP'' induced by T; T'' consists of the triangle $v_1v_2v_n$ and the fan of $(v_2, v_a, v_{a+1}, \ldots, v_n)$ centered at v_2. Because $w_1 \leq w_2 \leq w_a < w_n$, the fan of SP'' with center v_1 has lesser cost compared to T'', contradicting our assumption that T is optimal.

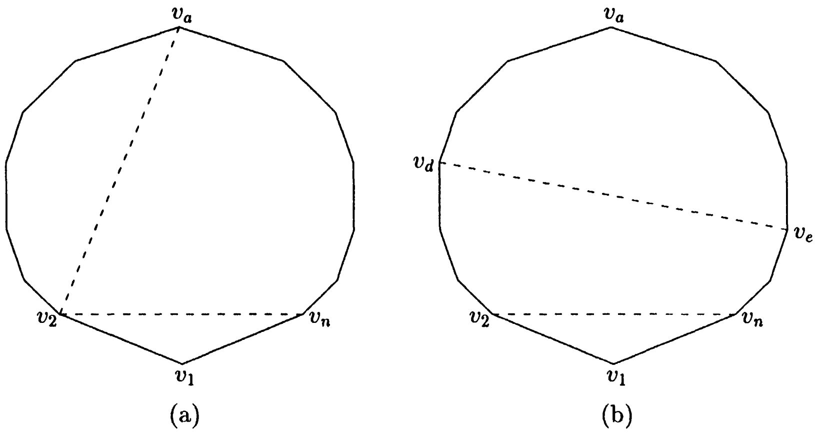


FIG. 1.

Case 2. v_2 is not a global minimum vertex in SP'. Let v_a, $3 \leq a < n$ be a global minimum vertex in SP'; we have $w_a < w_2 \leq w_n$. We consider two subcases.

Subcase 2.1. T' contains no ph-arcs of $SP' = (v_a, v_{a+1}, \ldots, v_n, v_2, v_3, \ldots, v_{a-1})$ (see Fig. 1a). By induction hypothesis, T' must be a fan of SP' with center v_a. Then, in T, the quadrilateral $v_1v_2v_av_n$ is triangulated by the arc v_2v_n. If we replace this arc by the arc v_1v_a, the resulting triangulation has lesser cost compared to T (because $w_1 \leq w_a < w_2 \leq w_n$), contradicting our assumption that T is optimal.

Subcase 2.2. T' contains some ph-arcs of $SP' = (v_a, v_{a+1}, \ldots, v_n, v_2, v_3, \ldots, v_{a-1})$ (see Fig. 1b). Let r be any such ph-arc. Because T contains no ph-arcs of SP, the upper subpolygon $SP'(r)$ of SP' must contain v_2 and v_n. Because this is true for any ph-arc r of SP' in T', these ph-arcs must be one above the other in SP'. Among these ph-arcs, let $r' = v_dv_e, d < e$, be the topmost one (i.e., the one closest to v_2v_n). Let SP'' denote $SP'(r') = (v_d, v_e, v_{e+1}, \ldots, v_n, v_2, v_3, \ldots, v_{d-1})$, and let T'' be the optimal triangulation of SP'' induced by T. Because r' is a ph-arc of SP', v_d, or v_e (whichever is of smaller weight) is a global minimum vertex in SP''. By our choice of r', T'' contains no ph-arcs of SP''. So, by induction hypothesis, T'' must be a fan of SP'' with center v_d or v_e. We will consider three subcases.

Subcase 2.2.1. $\{d, e\} \cap \{2, n\} = \phi$. Without loss of generality, let T'' be the fan of SP'' with center v_d. Then, in T, the quadrilateral $v_1v_2v_dv_n$ is triangulated by the arc v_2v_n. If we replace this arc by the arc v_1v_d, the resulting triangulation has lesser cost compared to T (because $w_1 \leq w_d \leq w_2 \leq w_n$, and $w_1 \leq w_a < w_2$), contradicting our assumption that T is optimal.

Subcase 2.2.2. $e = n$. Because $r' = v_d v_e$ is a ph-arc of SP', we have $w_d \le w_2 \le w_n$. Then, the proof is similar to that of Subcase 2.2.1.

Subcase 2.2.3. $d = 2$. If $w_e \le w_2$, then the proof is similar to that of Subcase 2.2.1, with v_e playing the role of v_d. If $w_e > w_2$, then we have $w_1 \le w_a < w_2 < w_e \le w_n$; then the proof is similar to that of Case 1. □

COROLLARY 2.1. *Let H be the set of h-arcs in an optimal triangulation of SP. If a subpolygon of SP that is bounded by some of the h-arcs in H contains none of these h-arcs in its interior, then a fan is an optimal triangulation of the subpolygon.*

Proof. Let $r = v_i v_j$ be an h-arc of SP. Then v_i or v_j (whichever is smaller weight) is a global minimum vertex in any subpolygon of SP that is bounded below by r. So, any ph-arc of the subpolygon is also a ph-arc of SP. Then the proof follows from Lemma 2. □

By Corollary 2.1, we only need to determine which of the h-arcs appear in an optimal triangulation of SP. For this, we need to be able to efficiently compute the cost of a fan of a subpolygon. For an h-arc $r = v_i v_j$, the *weight* of $SP(r)$, denoted by $W(r)$, is defined as follows. For *TPS*, $W(r) = (\mathit{weight}(r), \mathit{count}(r))$ where $\mathit{weight}(r)$ is the sum of weights of all the vertices in $SP(r)$ except v_i and v_j, and $\mathit{count}(r)$ is the number of vertices in $SP(r)$ excluding v_i and v_j; for *TPP*, $W(r) = \sum w_a w_b$ where the summation is taken over all the edges $v_a v_b$ of $SP(r)$ except r. The modality and the local minima of a polygon can be easily found in linear time. If we first find the local minima, then the One-Sweep Algorithm of Hu and Shing [4] can be easily modified so that it also computes $W(r)$ for each h-arc r. Note that the cost of a fan of a subpolygon can be easily computed using the $W(r)$s of the h-arcs r that bound the subpolygon.

The next concept we introduce is the *cutoff value* of an h-arc. For an h-arc $r = v_i v_j$, $i < j$, let $SP'(r)$ be the polygon obtained from $SP(r)$ by inserting a new vertex v between v_i and v_j, that is, $SP'(r) = (v, v_i, v_{i+1}, \dots, v_j)$ (see Fig. 2). The cutoff value of r, denoted by $co(r)$, is defined to be *the* value of $w(v)$ for which the following holds: the minimum cost of a triangulation of $SP'(r)$ that contains r equals the minimum cost of a triangulation of $SP'(r)$ that does not contain r; that is, there exists an optimal triangulation of $SP'(r)$ that contains r, and there exists another optimal triangulation of $SP'(r)$ that does not contain r (for *TPS*, such a value of $w(v)$ could be negative; for *TPP*, it is always nonnegative). Now, we prove that if such a value of $w(v)$ exists, then it is unique and is at most $\min(w_i, w_j)$. In §§3.1.1 and 3.1.2, we will show how to determine $co(r)$ (thereby also proving that is exists).

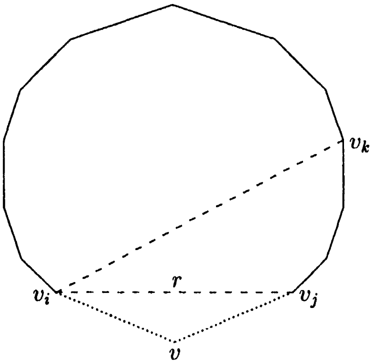


FIG. 2.

The motivation behind the definition of cutoff values is as follows. Let T be an optimal triangulation of SP; let r be an h-arc that is not in T. We say that an arc r' (*not* an h-arc of SP) in T *cuts* r if r' has one endpoint in $SP(; r)$ and another endpoint in $SP(r)$, with neither endpoint being an endpoint of r. By Corollary 2.1, all the arcs in T that cut r must have the same endpoint, say v, in $SP(;r)$. Also, by the definition of $co(r)$, we must have $w(v) \leq co(r)$.

LEMMA 3. *Let $SP'(r)$ be as defined above. If $w(v) > \min(w_i, w_j)$, then every optimal triangulation of $SP'(r)$ will contain r. So, if a value of $w(v)$ satisfies the definition of $co(r)$, then it is at most $\min(w_i, w_j)$.*

Proof. Without loss of generality, let $w_i \leq w_j$. Let $w(v) > w_i$, and let T' be any triangulation of $SP'(r)$ that does not contain r; we will prove that T' is not optimal. T' must contain a triangle vv_iv_k, for some k, $i < k < j$, and a triangulation of the polygon $(v, v_k, v_{k+1}, \ldots, v_j)$ (see Fig. 2). Consider replacing these two items by the triangle vv_iv_j and an optimal triangulation of the polygon $(v_i, v_k, v_{k+1}, \ldots, v_j)$. Because r is an h-arc, $w_k \geq w_j$; so, $c(w(v), w_i, w_k) \geq c(w(v), w_i, w_j)$. Also, because $w(v) > w_i$, the cost of an optimal triangulation of the polygon $(v, v_k, v_{k+1}, \ldots, v_j)$ is more that the cost of an optimal triangulation of the polygon $(v_i, v_k, v_{k+1}, \ldots, v_j)$. The last two statements together imply that T' is not optimal. □

LEMMA 4. *Let $SP'(r)$ be as defined above. For $w(v)$ equal to some fixed $w \leq \min(w_i, w_j)$, let T_1' be a triangulation that is of minimum cost among all triangulations of $SP'(r)$ that contain r (note that T_1' is independent of w). Let T_2' be a triangulation that is of minimum cost among all triangulations of $SP'(r)$ that do not contain r. Let $C_1'(x)$ and $C_2'(x)$ be the costs of (the* fixed *triangulations) T_1' and T_2', respectively, as a function of $w(v) = x$. Then $C_1'(x) - C_2'(x)$ increases strictly as x decreases. So, if $C_1'(w) \geq C_2'(w)$, then $C_1'(w') > C_2'(w')$ for all $w' < w$.*

Proof. For an h-arc r_p, let $T(r_p)$ be an optimal triangulation of $SP(r_p)$, and let $C(r_p)$ be its cost. T_1' consists of the triangle vv_iv_j and an optimal triangulation $T(r)$ of $SP(r)$. So

$$C_1'(x) = c(x, w_i, w_j) + C(r).$$

For $w(v)$ equal to (fixed) $w \leq \min(w_i, w_j)$, v is a global minimum vertex in $SP'(r)$. By Corollary 2.1, there exist h-arcs r_q, $1 \leq q \leq k$, no two of which are comparable, such that T_2' consists of r_q and $T(r_q)$, $1 \leq q \leq k$, and the fan of $(SP'(r))(; r_1, r_2, \ldots, r_k)$ centered at v (see Fig. 3) (if T_2' is the fan of $SP'(r)$ centered at v, we have $k = 0$). Let $F'(x)$ be the cost of this fan as a function of $x = w(v)$. Then

$$C_2'(x) = F'(x) + \sum_{q=1}^{k} C(r_q).$$

So, we have

$$C_1'(x) - C_2'(x) = C(r) - \sum_{q=1}^{k} C(r_q) - (F'(x) - c(x, w_i, w_j)).$$

Because r is an h-arc, the weight of each vertex in $SP(r)$ is at least $\max(w_i, w_j)$. It can be easily verified that, for both *TPS* and *TPP*, $F'(x) - c(x, w_i, w_j)$ is a strictly increasing function of x. So, the result follows. □

COROLLARY 4.1. *Let $SP'(r)$ be as defined above. There is at most one value of $w(v)$ that satisfies the definition of $co(r)$. Suppose that such a value exists; then it is at most* $\min(w_i, w_j)$. *Moreover, if $w(v) < co(r)$, then no optimal triangulation of $SP'(r)$ can contain r. If $w(v) > co(r)$, then every optimal triangulation of $SP'(r)$ will contain r. If $w(v) = co(r)$, then there exists an optimal triangulation of $SP'(r)$ that contains r, and there exists another optimal triangulation of $SP'(r)$ that does not contain r.*

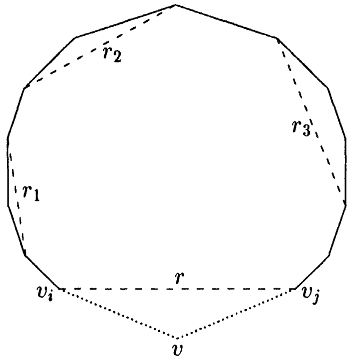


FIG. 3.

Proof. Follows from Lemmas 3 and 4. □

Hu and Shing [5] presented an $O(n \log n)$ algorithm for finding an optimal triangulation. In §3.1.1, we will describe their linear-time algorithm for unimodal polygons. In §3.1.2, we will describe its extension to an $O(n \log n)$ algorithm for general multimodal polygons. Our description of the algorithm is substantially simpler than the one given in [5]. The simplification is essentially due to the concept of cutoff values introduced above.

3.1.1. Algorithm for unimodal polygons. Let $SP = (v_1, v_2, \dots, v_n)$ be a unimodal polygon, where v_1 is the global minimum and v_a is the global maximum (see Fig. 4). From now onward, in our figures, we let the relative order of the y-coordinates of the vertices of a unimodal (sub)polygon be the same as the relative order of their weights. Each ph-arc of SP is of the form $v_i v_j$, where $1 < i < a < j$. In our figures, broken lines are used to represent h-arcs. Any two h-arcs of SP are comparable; so, the $n-3$ h-arcs are one above the other. Let the h-arcs be labeled $r_1, r_2, \dots, r_{n-3}$ from bottom to top. Let $r_i{}^1$ and $r_i{}^2$ be the two endpoints of r_i such that $w(r_i{}^1) \le w(r_i{}^2)$; $r_i{}^1$ and $r_i{}^2$ will be referred to as the *lower* and *upper* endpoints of r_i. Let $l \in \{2, n\}$ be such that $w_l = \min(w_2, w_n)$; $r_0 = v_1 v_l$ and $r_{n-2} = r^2_{n-3} v_a$ are considered to be *degenerate* h-arcs. For $1 \le i \le n-2$, r_i and r_{i-1} share the endpoint $r_i{}^1$; that is, $r_i{}^1$ is either r^1_{i-1} or r^2_{i-1}.

We now describe a linear-time algorithm for finding an optimal triangulation of SP. The algorithm performs a scan from top to bottom and processes the h-arcs one by one in the order $r_{n-2}, r_{n-3}, \dots, r_1, r_0$. When it processes r_i, it constructs an optimal triangulation $T(r_i)$ of $SP(r_i)$ and computes its cost $C(r_i)$ and $co(r_i)$. By Corollary 2.1, $T(r_i)$ can be represented by a list $L1(r_i)$ of the h-arcs that are in $T(r_i)$; $L1(r_i)$ contains these h-arcs in bottom to top order, and its first element is r_i. Note that the cutoff values of these arcs need not be in decreasing order. After processing r_i, the algorithm has two lists $L1(r_i)$ and $L2(r_i)$. $L2(r_i)$ is defined as follows. It is a sublist (i.e., a subsequence) of $L1(r_i)$. Its first element is r_i; an element of $L1(r_i)$ is in $L2(r_i)$ if and only if its cutoff value is less than that of all the preceding elements in $L1(r_i)$.

Now, we will show how to update $L1$ and $L2$ when the algorithm processes r_{i-1}. Let v' be the vertex in $\{r_{i-1}{}^1, r_{i-1}{}^2\} - \{r_i{}^1\}$. $L1(r_{i-1})$ is obtained from $L1(r_i)$ as follows. Remove the longest prefix of elements (i.e., h-arcs) all of which have cutoff values greater than or equal to $w(v')$, and then insert r_{i-1} at the front. Let r_j be the second element of $L1(r_{i-1})$. Then the triangulation $T(r_{i-1})$ consists of $T(r_j)$ and a fan of $SP(r_{i-1}; r_j)$ centered at $r_{i-1}{}^1$; its cost $C(r_{i-1})$ can be computed in constant time, using $C(r_j)$, $W(r_j)$ and $W(r_{i-1})$.

Before we can get $L2(r_{i-1})$, we need to compute $co(r_{i-1})$. We take $co(r_{n-2})$ to be $-\infty$ for *TPS*, and 0 for *TPP*. For *TPS*,

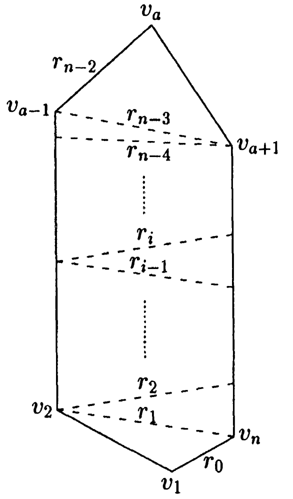


FIG. 4.

$$co(r_{n-3}) = w({r_{n-3}}^1) + w({r_{n-3}}^2) - w_a = w_{a-1} + w_{a+1} - w_a.$$

For *TPP*,

$$co(r_{n-3}) = w({r_{n-3}}^1)w({r_{n-3}}^2)w_a/[w_a(w({r_{n-3}}^1) + w({r_{n-3}}^2)) - w({r_{n-3}}^1)w({r_{n-3}}^2)]$$

$$= w_{a-1}w_aw_{a+1}/[w_a(w_{a-1} + w_{a+1}) - w_{a-1}w_{a+1}].$$

Before we can compute $co(r_{n-4})$, we need to know whether or not r_{n-3} will get "cutoff" at $co(r_{n-4})$, that is, whether $co(r_{n-4}) \leq co(r_{n-3})$ or $co(r_{n-4}) > co(r_{n-3})$.

For $i - 1 \leq n - 4$, $co(r_{i-1})$ is computed as follows. Recall, from Corollary 4.1, that $co(r_{i-1}) \leq w({r_{i-1}}^1)$. Let $L2'(r_{i-1})$ be the list obtained from $L2(r_i)$ by removing the longest prefix of elements all of which have cutoff values greater than to equal to $w({r_{i-1}}^1)$. We will first locate $co(r_{i-1})$ to be either equal to the cutoff value of a particular element in $L2'(r_{i-1})$, to be greater than the cutoff value of the first element in $L2'(r_{i-1})$, or to be strictly between the cutoff values of a particular pair of adjacent elements in $L2'(r_{i-1})$. In the latter two cases, $co(r_{i-1})$ is obtained by solving an equation.

Let $SP'(r_{i-1})$ be the polygon obtained from $SP(r_{i-1})$ by inserting a new vertex v between ${r_{i-1}}^1$ and ${r_{i-1}}^2$. Let $T'_1(r_{i-1})$ be the triangulation of $SP'(r_{i-1})$ that consists of $T(r_{i-1})$ and the triangle $v{r_{i-1}}^1{r_{i-1}}^2$. For any value of $w(v)$, $T'_1(r_{i-1})$ is of minimum cost among all triangulations of $SP'(r_{i-1})$ that contain r_{i-1}. Note that $T'_1(r_{i-1})$ is independent of $w(v)$. Its cost is

$$C'_1(r_{i-1}) = C(r_{i-1}) + c(w(v), w({r_{i-1}}^1), w({r_{i-1}}^2)).$$

Let $T'_2(r_{i-1})$ denote a triangulation that is of minimum cost among all triangulations of $SP'(r_{i-1})$ that do not contain r_{i-1}. Clearly, $T'_2(r_{i-1})$ depends on $w(v)$.

We will let $w(v)$ successively take the cutoff value of each of the h-arcs in $L2'(r_{i-1})$. Suppose that we are currently considering $co(r_p)$ as the value of $w(v)$, for some $r_p \in L2'(r_{i-1})$; let r_q be the h-arc following r_p in $L2'(r_{i-1})$. Let $T'_{2,p}(r_{i-1})$ be the triangulation of $SP'(r_{i-1})$ that consists of r_q, $T(r_q)$ and the fan of $(SP'(r_{i-1}))(; r_q)$ centered at v. By Corollary 4.1, when $co(r_q) < w(v) \leq co(r_p)$, $T'_{2,p}(r_{i-1})$ is of minimum cost among all triangulations of $SP'(r_{i-1})$

that do not contain r_{i-1}; its cost $C'_{2,p}(r_{i-1})$ can be computed (as a function of $w(v)$) in constant time, using $C(r_q)$, $W(r_q)$ and $W(r_{i-1})$. For $co(r_q) < w(v) \leq co(r_p)$, we take $T'_2(r_{i-1})$ to be $T'_{2,p}(r_{i-1})$; its cost $C'_2(r_{i-1})$ equals $C'_{2,p}(r_{i-1})$. We compare $C'_2(r_{i-1})$ with $C'_1(r_{i-1})$, at $w(v) = co(r_p)$.

This process is repeated for each $r_p \in L2'(r_{i-1})$ sequentially from the front until, for the first time, $C'_2(r_{i-1}) \leq C'_1(r_{i-1})$ at $w(v) = co(r_p)$. If the two costs are equal, then $co(r_{i-1}) = co(r_p)$; else, $co(r_{i-1})$ is greater than $co(r_p)$, but less than the cutoff values of all the arcs that preceed r_p in $L2'(r_{i-1})$. In the latter case, by Corollary 4.1, r_p would be the first (i.e., bottom most) h-arc in $T'_2(r_{i-1})$, when $w(v)$ is arbitrarily close to $co(r_{i-1})$. Then $co(r_{i-1})$ can be computed in constant time, as the value of $w(v)$ that satisfies the equation $C'_2(r_{i-1}) = C'_1(r_{i-1})$ (for *TPS*, this equation is linear in $w(v)$ and $\vec{w}$; for *TPP*, it is linear in $w(v)$ but cubic in $\vec{w}$).

LEMMA 5. *The algorithm outlined above correctly computes* $co(r_{i-1})$.

Proof. Because $T'_1(r_{i-1})$ is a fixed triangulation, $C'_1(r_{i-1})$ is a continuous function of $w(v)$. For a fixed $r_p \in L2'(r_{i-1})$, $T'_{2,p}(r_{i-1})$ is a fixed triangulation; so, $C'_{2,p}(r_{i-1})$ is a continuous function of $w(v)$. By the definition of $co(r_p)$, $C'_2(r_{i-1})$ is continuous at $w(v) = co(r_p)$, for each $r_p \in L2'(r_{i-1})$. So, $C'_2(r_{i-1})$ is a continuous function of $w(v)$. Hence, $C'_2(r_{i-1}) - C'_1(r_{i-1})$ is a continuous function of $w(v)$. At $w(v) = w({r_{i-1}}^1)$, by Lemma 3, $C'_2(r_{i-1}) - C'_1(r_{i-1})$ is nonnegative. As $w(v)$ decreases and approaches its limiting value of $co(r_{n-2})$ (i.e., $-\infty$ for *TPS*, 0 for *TPP*), $C'_2(r_{i-1}) - C'_1(r_{i-1})$ becomes negative. So, there must be a value of $w(v)$ for which $C'_2(r_{i-1}) = C'_1(r_{i-1})$; by definition, this value is $co(r_{i-1})$. It is clear that the above algorithm correctly determines this value. □

Finally, $L2(r_{i-1})$ is obtained from $L2'(r_{i-1})$ as follows. Remove the longest prefix of elements all of which have cutoff values greater than or equal to $co(r_{i-1})$ and then insert r_{i-1} at the front.

$L1(r_0)$ contains the h-arcs in an optimal triangulation $T(r_0)$ of $SP(r_0) = SP$, and $C(r_0)$ is the cost of $T(r_0)$. Obtaining $L1(r_{i-1})$ and $L2(r_{i-1})$ takes time proportional to the number of elements removed from $L1(r_i)$ and $L2(r_i)$. Hence the above algorithm runs in linear time. Note that the algorithm can compute $C(r_{i-1})$, $co(r_{i-1})$, and $L2(r_{i-1})$ from $L2(r_i)$, without using $L1(r_i)$. So we can compute $co(r_i)$ and $L2(r_i)$ for i varying from $n-3$ down to 0, in linear time, without keeping track of $L1$. After the cutoff values of all the h-arcs have been found, we can determine $L1(r_0)$ in linear time by performing a bottom to top scan of SP. Also note that before terminating, the algorithm computes $co(r_0)$. Both these observations will be of use in the next subsection.

3.1.2. Algorithm for multimodal polygons. In this subsection, we describe the $O(n \log n)$ algorithm for general multimodal polygons. First, we describe the algorithm for bimodal polygons. Let $SP = (v_1, v_2, \ldots, v_n)$ be a bimodal polygon, where v_1 is the global minimum, v_b is a local minimum, and v_{a1} and v_{a2} are the local maxima, $1 < a1 < b < a2 < n$, and $w_n < w_b$ (see Fig. 5). Then there exists a vertex v_d, $1 \leq d < a1$, such that $w_d < w_b \leq w_{d+1}$; also there exists a vertex v_e, $a2 < e \leq n$, such that $w_e < w_b \leq w_{e-1}$. $r_{1,0} = v_d v_b$, $r_{2,0} = v_b v_e$, and $r_0 = v_d v_e$ are h-arcs of SP (if $d = 1$ and $e = n$, then $r_0 = v_1 v_n$ is a degenerate h-arc). $SP(r_{1,0})$, $SP(r_{2,0})$, and $SP(; r_0)$ are unimodal polygons. The h-arcs of $SP(; r_0)$ will be labeled as $r_1, r_2, \cdots$ from top to bottom. Without loss of generality, let $w_d \leq w_e$.

The definitions of r^1 and r^2 (the lower and upper endpoints of r), $T(r)$, and $C(r)$ for an h-arc r of a multimodal polygon are the same as those in §3.1.1 for unimodal polygons. Note that once the cutoff values of all the h-arcs have been found, we can determine the h-arcs in an optimal triangulation of SP by performing a bottom to top scan in linear time.

The algorithm for finding the cutoff values for the above bimodal polygon SP works as follows. First compute the lists $L12(r_{1,0})$ and $L22(r_{2,0})$ (analogous to the list $L2$ in §3.1.1) for

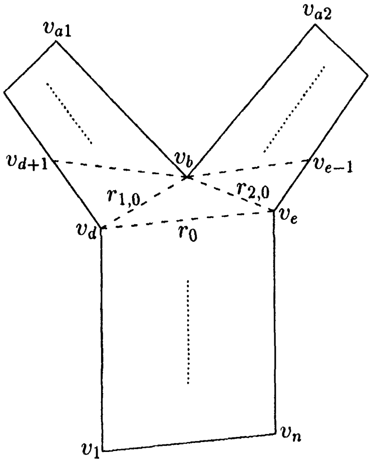


FIG. 5.

unimodal subpolygons $SP(r_{1,0})$ and $SP(r_{2,0})$, respectively, using the algorithm in §3.1.1. Because $w_d \leq w_e$, let $L22'(r_0)$ be the list obtained from $L22(r_{2,0})$ by removing the longest prefix of elements all of which have cutoff values greater than or equal to w_d. If $r_{2,i}$ is the first element of $L22'(r_0)$, then $T(r_0)$ consists of $T(r_{1,0})$, $T(r_{2,i})$, and a fan of $SP(r_0; r_{1,0}, r_{2,i})$ centered at v_d; its cost $C(r_o)$ can be computed in constant time, using $C(r_{1,0})$, $C(r_{2,i})$, $W(r_0)$, $W(r_{1,0})$, and $W(r_{2,i})$. Let $L2'(r_0)$ be the list obtained by *merging* $L12(r_{1,0})$ and $L22'(r_0)$ into a single list such that the cutoff values of the elements in $L2'(r_0)$ are in decreasing order from left to right. Using $L2'(r_0)$, we can compute $co(r_0)$ as explained in §3.1.1. Then $L2(r_0)$ is obtained from $L2'(r_0)$ as follows. Remove the longest prefix of elements all of which have cutoff values greater than or equal to $co(r_0)$ and then insert r_0 at the front. Then, we can perform a top to bottom scan of $SP(; r_0)$; for $i \geq 1$, $co(r_i)$, and $L2(r_i)$ can be computed from $L2(r_{i-1})$ as explained in §3.1.1. This completes our description of the algorithm for bimodal polygons.

In the above bimodal polygon, if $r_{1,i}$ and $r_{2,j}$ are h-arcs of $SP(r_{1,0})$ and $SP(r_{2,0})$, respectively, then they are incomparable, but they are above r_0. The h-arc r_0 will be called a *bridge*. In general, a *bridge* is an h-arc r whose endpoints are the lower endpoints of two other h-arcs r' and r''; r' and r'' will have the same upper endpoint. The preceding paragraph describes a general procedure for obtaining $L2(r)$ from $L2(r')$ and $L2(r'')$. In general, an m-modal polygon will have $m - 1$ bridges. It is easily seen how the above top-down algorithm for bimodal polygons can be extended to general m-modal polygons. Unfortunately, we have no control over the positions of the bridges or the relative order of the sizes of the two lists to be merged. If we actually form the sorted list $L2(r)$ for each bridge r, we cannot guarantee $O(n \log n)$ time complexity. Note that the only operations we need to perform on the $L2$ lists are the following:

INSERT: Insert an element into the list

DELETE: Delete an element from the list

MAX: Find the largest element in the list

UNION: Take the union of two lists

In fact, we only need to INSERT an element that is larger than all the other elements in the list and to DELETE the largest element from the list. When we process an h-arc r, we will maintain the unordered set of elements in $L2(r)$ as a *Mergeable Heap* (see [1]). This data structure allows each INSERT, DELETE, MAX, or UNION operation to be performed in $O(\log n)$ time. Because we only need to perform $O(n)$ such operations, the overall algorithm runs in $O(n \log n)$ time. Note that apart from the heap operations mentioned above, all the other operations performed by the algorithm require only linear time.

3.2. The lower bound. The algorithm in §3.1 finds an optimal triangulation of $SP(r)$ for each h-arc r. This seems wasteful, because we are only interested in an optimal triangulation of SP. The natural question is whether it is possible to reduce the heap operations and find an optimal triangulation of SP in $o(n \log n)$ time. We will prove that this is not possible for *TPS*. We will show that there are $n^{\Omega(n)}$ ross's for *TPS* and *TPP*. Then a tight $\Omega(n \log n)$ lower bound for *TPS* will follow from Theorem 3. If we could extend our lower bound technique to bounded degree algebraic decision trees, we would have a tight $\Omega(n \log n)$ lower bound for *TPP*.

Consider the following worst-case instance of the algorithm described in §3.1. Let m be even, and let $SP = (p_m, p_{m-2}, \ldots, p_2, p_0, q_{1,1}, q_{1,2}, q_{2,1}, q_{2,2}, \ldots, q_{m,1}, q_{m,2}, p_1, p_3, \ldots, p_{m-1})$ (see Fig. 6 for $m = 4$). SP has $n = 3m + 1$ vertices. Let $q_{0,2} = p_0$, $p_{-1} = q_{m,2}$ and $p_{-2} = q_{1,2}$. Let $w(p_i) > w(p_{i+1})$, $-2 \le i < m$, $w(q_{1,1}) > w(q_{1,2})$, and $w(q_{j,1}) > w(q_{j-1,2}) > w(q_{j,2})$, $1 < j \le m$. Then SP is m-modal, p_m is the global minimum, $q_{j,2}$, $1 \le j < m$ are the local minima, and $q_{j,1}$, $1 \le j \le m$, are the local maxima.

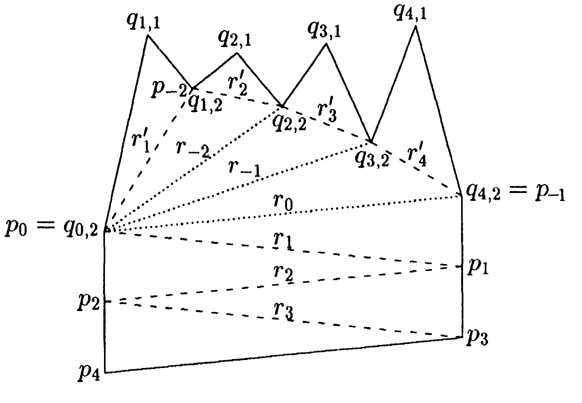


FIG. 6.

$r_i = p_i p_{i-1}$, $1 \le i < m$; $r_i = p_0 q_{m+i,2}$, $-(m-2) \le i \le 0$; $r'_j = q_{j-1,2} q_{j,2}$, $1 \le j \le m$; these are the h-arcs. $r_i = p_0 q_{m+i,2}$, $-(m-2) \le i \le 0$, are the bridges. $r_m = p_m p_{m-1}$ is a degenerate h-arc.

Let $L = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ be the list of r'_j, $1 \le j \le m$, in decreasing order of cutoff values. $\alpha_j = r'_{\pi(j)}$, $1 \le j \le m$, for some permutation π of $(1, 2, \ldots, m)$; we let L be denoted by L_π. We will show that, for both *TPS* and *TPP*, all the $m! = [(n-1)/3]!$ permutations π are possible. We will also show that each π corresponds to a ross, and that no two π's correspond to the same ross. This will imply that there are at least $m!$ ross's for *TPS* and *TPP*. We have the following.

LEMMA 6. *Let π be any fixed permutation of $(1, 2, \ldots, m)$. Suppose that the weights $\vec{w} = (w_1, w_2, \ldots, w_n)$ of the vertices of SP are such that*

(i) *$w(q_{1,2}) - w(q_{m-1,2})$ is arbitrarily small,*

(ii) *$L_\pi = (\alpha_1, \alpha_2, \ldots, \alpha_m)$ is the list of r'_js in decreasing order of cutoff values,*

(iii) *$w(p_i) < co(\alpha_i) < w(p_{i-1})$ for $1 \le i \le m$, and*

(iv) *$co(r_{i-1}) = w(p_i)$ for $1 \le i \le m$.*

Let $\hat{T}$ *be the triangulation of* SP *consisting only of h-arcs (see Fig. 6). For* $0 \le k \le m$, *let* T_k *be the triangulation obtained from* $\hat{T}$ *as follows. Remove* α_j, $1 \le j \le k$, *and* r_i, $-(m-2) \le i < k$; *the resulting figure will have only one nontriangular face* F_k, *and* p_k *will be the smallest weight vertex in that face; triangulate that face as a fan. Note that* $T_0 \equiv \hat{T}$. *Let* $S_\pi = \{T_k | 0 \le k \le m\}$. *The set* $sol(\vec{w})$ *of all optimal triangulations of* SP *is* S_π; *so* S_π *is a* ross.

Proof. Let π and $\vec{w}$ be such that conditions (i)–(iv) hold. By (i), for both *TPS* and *TPP*, the cutoff values of the top $m-2$ bridges $r_i = p_o q_{m+i,2}$, $-(m-2) \le i < 0$, which are all less than $w(p_0)$, are very close to $w(p_0)$. Because by (iv), $co(r_0) = w(p_1) < w(p_0)$, if an optimal triangulation does not contain the bridge r_0, then it cannot contain any of the other bridges. Also by (iv), if an optimal triangulation does not contain r_k for some k, $1 \le k < m$, then it cannot contain any of the h-arcs r_i, $0 \le i < k$.

For an optimal triangulation T of SP, let k be the smallest integer, $0 \le k \le m$, such that T contains r_k. Then, by the arguments in the preceding paragraph and by (iii), T must be T_k. Also, it is easily seen that each of the triangulations T_k, $0 \le k \le m$, is optimal. Hence $S_\pi = \text{sol}(\vec{w})$ is a ross. □

Note that π serves as a "signature" for S_π. Hence two different π's cannot correspond to the same S_π. In §3.2.1, we will show that, for *TPS*, for each permutation π of $(1, 2, \dots, m)$, we can find a $\vec{w}$ that satisfies the conditions of Lemma 6. In §3.2.2, we will prove the analogous result for *TPP*.

3.2.1. Tight lower bound for TPS. Let π be a fixed permutation of $(1, 2, \dots, m)$. We will find a $\vec{w}$ that satisfies the conditions of Lemma 6. Let ${\alpha_i}^1$ and ${\alpha_i}^2$ denote the lower and upper endpoints of α_i, respectively; let u_i denote the unique vertex of SP that lies between ${\alpha_i}^1$ and ${\alpha_i}^2$. Condition (iii) of Lemma 6 translates to the following: for $1 \le i \le m$,

$$(1) \qquad w(p_i) + w(u_i) < w({\alpha_i}^1) + w({\alpha_i}^2) < w(p_{i-1}) + w(u_i).$$

Condition (iv) of the lemma essentially equates the costs of T_{i-1} and T_i, for $1 \le i \le m$. T_{i-1} and T_i differ only in the triangulation of the face F_i. In T_i, the triangulation of F_i is a fan centered at p_i. In T_{i-1}, the triangulation of F_i consists of a fan of F_{i-1} centered at p_{i-1}, and the triangles ${\alpha_i}^1{\alpha_i}^2 u_i$ and $p_i p_{i-1} p_{i-2}$. So, condition (iv) translates to the following: for $1 \le i \le m$,

$$(2) \; (m+2i-3)w(p_{i-1}) + w(p_{i-2}) + w({\alpha_i}^1) + w({\alpha_i}^2) = (m+2i-2)w(p_i) + w(p_{i-3}) + w(u_i).$$

Solving (2) for $w(u_i)$, we get

(2′)

$$w(u_i) = w({\alpha_i}^1) + w({\alpha_i}^2) + (m+2i-3)w(p_{i-1}) - (m+2i-2)w(p_i) + w(p_{i-2}) - w(p_{i-3}).$$

Substituting (2′) in (1), we get

$$(1') \; (m+2i-3)(w(p_{i-1}) - w(p_i)) < w(p_{i-3}) - w(p_{i-2}) < (m+2i-2)(w(p_{i-1}) - w(p_i)).$$

We can first choose $w(p_i)$, for i varying from m down to -2, obeying (1′). Note that (1′) and (2′) do not restrict $w(q_{j,2})$, $1 < j < m$; so, we can choose them such that condition (i) of Lemma 6 is satisfied, and $w(q_{j-1,2}) > w(q_{j,2})$ for $1 < j \le m$. Finally, we can compute the $w(u_i)$'s (i.e., $w(q_{j,1})$'s) from (2′). Then, from (1′) and (2′), we are guaranteed that

$$w({\alpha_i}^1) + w({\alpha_i}^2) - w(p_{i-1}) < w(u_i) < w({\alpha_i}^1) + w({\alpha_i}^2) - w(p_i)$$

or in particular that $w(u_i) > w({\alpha_i}^2)$. So we are guaranteed that $w(q_{1,1}) > w(q_{1,2})$ and $w(q_{j,1}) > w(q_{j-1,2})$ for $1 < j \le m$. Hence we have shown that for each π, we can find a $\vec{w}$ that satisfies the conditions of Lemma 6. By Theorem 4, this leads to the following.

THEOREM 5. *Any linear decision tree that solves TPS_I has $\Omega(n \log n)$ height.*

Note that by (1′), $[w(p_{-2}) - w(p_{-1})]/[w(p_{m-1}) - w(p_m)]$ is of the order $m^{\theta(m)}$. So our lower bound proof requires that the input consist of a large number of bits.

3.2.2. Ross count for TPP. Let π be a fixed permutation of $(1, 2, \dots, m)$. Let ${\alpha_i}^1$, ${\alpha_i}^2$, and u_i be as defined in the preceding subsection. To simplify the arithmetic, we change condition (iii) of Lemma 6 to

$$w(p_i) = co(\alpha_i) < w(p_{i-1}) \quad \text{for} \quad 1 \le i \le m.$$

Since $w(p_i) < w(p_{i-1})$, this is equivalent to replacing (iii) by

$$\text{(iii}'\text{)} \qquad w(p_i) = co(\alpha_i) \quad \text{for} \quad 1 \le i \le m.$$

For $1 \le k \le m$, let T'_k be the triangulation of SP obtained from T_k by replacing the arc $p_k u_k$ by the h-arc α_k. Let $S'_\pi = S_\pi \cup \{T'_k | 1 \le k \le m\}$. Let Lemma 6′ be the statement that results from Lemma 6 when condition (iii) is replaced by (iii'), and S_π is replaced by S'_π. Proof of Lemma 6′ is similar to that of Lemma 6.

We will find a $\vec{w}$ that satisfies the conditions of Lemma 6′. Condition (iii′) of Lemma 6′ translates to the following: for $1 \le i \le m$,

$$(3) \qquad w(p_i)w(u_i)(w({\alpha_i}^1) + w({\alpha_i}^2)) = w({\alpha_i}^1)w({\alpha_i}^2)(w(p_i) + w(u_i)).$$

Solving (3) for $w(u_i)$, we get

$$(3') \qquad 1/w(u_i) = 1/w({\alpha_i}^2) + [1/w({\alpha_i}^1) - 1/w(p_i)].$$

Because $w({\alpha_i}^1) > w(p_i)$, the term in the square bracket is negative. So, as long as the right-hand side is positive, $w(u_i) > w({\alpha_i}^2)$; that is, $w(q_{1,1}) > w(q_{1,2})$ and $w(q_{j,1}) > w(q_{j-1,2})$ for $1 < j \le m$.

For $0 \le i \le m$, let $W'(F_i)$ denote $\sum w(v)w(v')$ where the summation is taken over all the edges vv' around the face F_i (see Lemma 6 for the definition of F_k), except the edges $p_i p_{i-1}$ and $p_i p_{i-2}$. Condition (iv) of the lemma essentially equates the costs of T_{i-1} and T_i, for $1 \le i \le m$. T_{i-1} and T_i differ only in the triangulation of the face F_i. In T_i, the triangulation of F_i is a fan centered at p_i. In T_{i-1}, the triangulation of F_i consists of a fan of F_{i-1} centered at p_{i-1} and the triangles ${\alpha_i}^1{\alpha_i}^2 u_i$ and $p_i p_{i-1} p_{i-2}$. Because of condition (iii′), condition (iv) translates to the following: for $1 \le i \le m$,

$$(4) \quad w(p_i)w(p_{i-1})w(p_{i-2}) + w(p_{i-1})W'(F_{i-1}) = w(p_i)w(p_{i-1})w(p_{i-3}) + w(p_i)W'(F_{i-1}).$$

Solving (4) for $w(p_i)$, we get

$$(4') \qquad 1/w(p_i) = 1/w(p_{i-1}) + [w(p_{i-3}) - w(p_{i-2})]/W'(F_{i-1}).$$

This guarantees that $w(p_i) < w(p_{i-1})$. We can first choose $w(p_{-2}) = w(q_{1,2})$, $w(q_{j,2})$, $1 < j < m$, $w(q_{m,2}) = w(p_{-1})$, and $w(p_0)$. Then we can compute the pair $(w(p_i), w(u_i))$ from (4′) and (3′), for i varying from 1 to m.

Because $W'(F_{i-1}) \ge w(p_{i-2})w(p_{i-3})$, (4′) implies that

$$(5) \qquad 1/w(p_i) - 1/w(p_{i-1}) \le 1/w(p_{i-2}) - 1/w(p_{i-3}).$$

By taking $w(p_{-2}) - w(p_0)$ to be small, say $1/w(p_0) - 1/w(p_{-2}) = 1/(2mw(p_0))$, we can guarantee, using (5) and a simple induction, that

$$1/w(p_m) \le (3/2)(1/w(p_0)) \le 2/w(p_{-2}).$$

Then it follows that the right-hand side of (3′) is positive for all i, $1 \leq i \leq m$. Note that (3′) and (4′) do not restrict $w(q_{j,2})$, $1 < j < m$; so, we can choose them such that condition (i) of Lemma 6′ is satisfied, and $w(q_{j-1,2}) > w(q_{j,2})$ for $1 < j \leq m$. Hence, we have shown that for each π, we can find a $\vec{w}$ that satisfies the conditions of Lemma 6′. This leads to the following.

LEMMA 7. $|\mathrm{ROSS}(TPP)| \geq [(n-1)/3]!$.

3.3. The relationship between TPS and TPP. The definitions of *TPS* and *TPP* are very similar. In §3.1, we saw that the two problems can be solved by very similar algorithms. In §3.2, we saw that we can obtain a good lower bound on the number of ross's for the two problems using very similar ideas. In this subsection, we will discuss an even deeper relationship between the two problems. For an instance $\vec{w} = (w_1, w_2, \ldots, w_n)$ and a triangulation T of SP, let $c_T{}^{TPS}(\vec{w})$ and $c_T{}^{TPP}(\vec{w})$ be the costs of T in *TPS* and *TPP*, respectively. Let $\mathrm{sol}^{TPS}(\vec{w})$ and $\mathrm{sol}^{TPP}(\vec{w})$ denote the sets of all optimal solutions for $\vec{w}$ in *TPS* and *TPP*, respectively. For any real number N, let $\vec{N}$ denote $(N, N, \ldots, N) \in R^n$.

LEMMA 8. *For any $\vec{w}$, there exists a positive real number $N_{\vec{w}}$ such that* $\mathrm{sol}^{TPS}(\vec{w}) \supseteq \mathrm{sol}^{TPP}(\vec{w} + \vec{N})$, *for all $N \geq N_{\vec{w}}$.*

Proof. For a triangulation T of SP and fixed $\vec{w}$,

$$c_T{}^{TPP}(\vec{w} + \vec{N}) = (n-2)N^3 + N^2 c_T{}^{TPS}(\vec{w}) + O(N).$$

(Note that the hidden constant in the $O(N)$ term is a function of $\vec{w}$.) Hence, for any two triangulations T and T' of SP, and for all sufficiently large N, if $c_T{}^{TPP}(\vec{w}+\vec{N}) \leq c_{T'}{}^{TPP}(\vec{w}+\vec{N})$ then $c_T{}^{TPS}(\vec{w}) \leq c_{T'}{}^{TPS}(\vec{w})$. Hence the result follows. □

For a polynomial $f(\vec{w})$, let $\psi f(\vec{w})$ be the coefficient of the highest power of N in $f(\vec{w} + \vec{N})$; note that $\psi f(\vec{w})$ is a polynomial in $\vec{w}$, and its degree is no more than that of $f(\vec{w})$. For an algebraic decision tree D, let $\psi(D)$ be the decision tree obtained from D by replacing at each internal node v, the comparison $f_v(\vec{w}) : 0$ by the comparison $\psi f_v(\vec{w}) : 0$. The node in $\psi(D)$ corresponding to $v \in D$ will be referred to as $\psi(v)$. We have the following.

LEMMA 9. *Let D be an algebraic decision tree that solves TPP. For $\vec{w} \in R^n$, if $\vec{w} \in$* $\mathrm{set}(\psi(l))$ *for some $l \in L'(D)$, then* $\mathrm{sol}(\psi(l)) \in \mathrm{sol}^{TPS}(\vec{w})$; *so there exists a set $A \subseteq R^n$ of measure zero, such that $\psi(D)$ correctly solves TPS for any $\vec{w} \in R^n - A$.*

Proof. Let $\vec{w} \in \mathrm{set}(\psi(l))$ for some $l \in L'(D)$. Then, for all sufficiently large N, we have

$$\mathrm{sign}(f_v(\vec{w} + \vec{N})) = \mathrm{sign}(\psi f_v(\vec{w})) \neq 0$$

for all $v \in \xi_l$; so $\vec{w} + \vec{N} \in \mathrm{set}(l)$. Because D correctly solves *TPP*, $\mathrm{sol}(l) \in \mathrm{sol}^{TPP}(\vec{w} + \vec{N})$. By Lemma 8 we have that $\mathrm{sol}(\psi(l)) = \mathrm{sol}(l) \in \mathrm{sol}^{TPS}(\vec{w})$.

Finally, let $A = \cup_{l \in L(D)-L'(D)} \mathrm{set}(\psi(l))$; $A \subseteq \{\vec{w} \in R^n | \psi f_v(\vec{w}) = 0 \text{ for some } v \in D\}$, and so A is of measure zero. $\psi(D)$ correctly solves *TPS* for any $\vec{w} \in R^n - A$. □

This leads to the following.

LEMMA 10. *Let H_{TPS} be a lower bound on the height of any degree d algebraic decision tree that solves TPS for all but a measure zero set of inputs. Then H_{TPS} is also a lower bound on the height of any degree d algebraic decision tree that solves TPP.*

4. Conclusion. We first presented a new technique for obtaining lower bounds on the worst-case time-complexity of optimization problems, in the linear decision tree model of computation. Then, we used this technique to obtain a tight $\Omega(n \log n)$ lower bound for a problem of finding a minimum cost triangulation of a convex polygon with weighted vertices. This problem is very similar to the problem of finding an optimal order of computing a matrix chain product. If we could extend our lower bound technique to bounded degree algebraic decision trees, we would have a tight $\Omega(n \log n)$ lower bound for this latter problem.

Acknowledgment. The author is very grateful to the referee, Prof. George Lueker, for many valuable suggestions. In particular, the results in §3.3 were suggested by him.

REFERENCES

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

[2] M. BEN-OR, *Lower bounds for algebraic computation trees*, Proc. 15th Ann. ACM Symp. on Theory of Comput., May 1983, pp. 80–86.

[3] M. GARDNER, *Catalan numbers*, Scientific American, (234) June 1976, pp. 120–124.

[4] T. C. HU AND M. T. SHING, *Computation of matrix chain products, Part* I, SIAM J. Comput., 11 (1982), pp. 362–373.

[5] ———, *Computation of matrix chain products, part* II, SIAM J. Comput., 13 (1984), pp. 228–251.

[6] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, New Jersey, 1982.

[7] A. C. YAO AND R. L. RIVEST, *On the polyhedral decision problem*, SIAM J. Comput., 9 (1980), pp. 343–347.

[8] A. C. YAO, *Lower bounds for algebraic computation trees with integer inputs*, SIAM J. Comput., 20 (1991), pp. 655–668.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 852–863, August 1994

ON "AXIOMATISING FINITE CONCURRENT PROCESSES"*

LUCA ACETO[†]

Abstract. In his pioneering paper [*Axiomatising finite concurrent processes*, SIAM J. Comput., 17 (1988), pp. 997–1017], Hennessy gave complete axiomatizations of Milner's observational congruence and of t-observational congruence which made use of an auxiliary operation to axiomatize parallel composition. Unfortunately, those axiomatizations turn out to be flawed due to the subtle interplay between Hennessy's auxiliary parallel operator and synchronization. The aim of this paper is to present correct versions of the equational characterizations given in Hennessy's paper. Some of the problems which arise in giving operational semantics to the auxiliary operators used by Bergstra and Klop and Hennessy in the theory of congruences like Milner's observational congruence are also discussed.

Key words. concurrent processes, observational congruence, t-observational congruence, equational logic

AMS subject classification. 68Q55

1. Introduction.

In his seminal paper [14], Matthew Hennessy gave complete axiomatizations of two behavioural congruences, namely, those associated with Milner's *weak bisimulation equivalence* [19] and *t-observational equivalence* [14] (also known as *split*-2 *equivalence* [10] and *timed equivalence* [1]), over a simple language for concurrent processes. Hennessy's paper [14] evolved from an early preprint, entitled *On the relationship between time and interleaving*, which dated back to 1981 and, in my opinion at least, did not receive the attention it deserved at the time of its first circulation.

Hennessy's *On the relationship between time and interleaving* and its published version [14] have historically played an important role in the development of the theory of process algebras for at least two reasons. First, the equational characterization of observational congruence presented in these papers was, to the best of my knowledge, the first one to use auxiliary operators in the axiomatization of CCS parallel composition [19]. At more or less the same time, J. A. Bergstra and J. W. Klop were working on a finite axiomatization of strong bisimulation equivalence over ACP which used two auxiliary operators [4], but extensions of their ideas to a setting involving internal actions were first presented in [6]. Secondly, Hennessy's papers presented the first axiomatization known to the author of a noninterleaving behavioural equivalence and its laws have helped shape the form of many axiomatizations which followed. (See, e.g., [7], [8], [18], [15].)

Unfortunately, however, there are subtle problems with the axiomatizations published in [14]. In particular, two of the axioms given by Hennessy for his auxiliary parallel operation are *unsound* due to the problems introduced by synchronization. In fact, the whole issue of giving semantics to the auxiliary operations used in [4], [6], [14] to axiomatize various parallel composition operators turns out to be rather subtle in the theory of behavioural congruences associated with weak bisimulation-like equivalences, such as observational congruence and t-observational congruence. The aim of this paper is to present correct versions of the axiomatizations given in [14]. In passing, I shall also comment on some of the issues involved in giving suitable operational semantics for the auxiliary operations of ACP in the setting of observational congruence and related congruences. I hope that this will make this paper a useful reference for researchers interested in complete axiomatizations of behavioural congruences.

The paper is organized as follows: §2 is devoted to a discussion of Hennessy's axiomatization of t-observational congruence [14] and an example showing that it is unsound is

*Received by the editors Janaury 12, 1993; accepted for publication (in revised form) May 18, 1993.

[†]School of Cognitive and Computing Studies, University of Sussex, Falmer, Brighton BN1 9QH, United Kingdom (luca@cogs.sussex.ac.uk).

given. I then present a sound and complete axiomatization for this congruence which uses the auxiliary operations of ACP. In §2.1, I present an example showing that the axiomatization of observational congruence in [14] is also unsound and give a correct sound and complete axiomatization for this congruence. Finally, in §3, I discuss the operational semantics of the auxiliary operations of ACP in the setting of observational congruence.

2. An axiomatization of Hennessy's t-observational congruence. I assume that the reader is familiar with [14] and the basic notions on process algebras and bisimulation equivalence. The uninitiated reader is referred to the textbooks [19], [3] for extensive motivations and background. As this is not an introductory paper, I shall feel free to refer the reader to the motivations, definitions, and results given in [14]. Precise pointers to material in [14] will be given whenever necessary.

The language **P** used by Hennessy in [14] is a simple extension of finite, restriction- and relabelling-free CCS. It is given by the grammar

$$p ::= \mathbf{0} \mid \mu.p \mid p + p \mid p \| p \mid p \lfloor p$$

where μ ranges over the set of *actions* **Act**. The set **Act** is assumed to have the form $\{\tau\} \cup \Lambda \cup \bar{\Lambda}$, where Λ is a given countable set of names, $\bar{\Lambda} = \{\bar{a} \mid a \in \Lambda\}$ is the set of complement names, and τ is a distinguished action. As usual, we assume that complementation is symmetric, i.e., $\bar{\bar{a}} = a$. We use **VAct** to denote $\Lambda \cup \bar{\Lambda}$, the set of *visible actions*, and a, b to range over it.

The operational semantics for the language **P** given by Hennessy in §2.1 of [14] is based on the idea that visible actions have a beginning and an ending. Moreover, these distinct events may be observed and are denoted by $S(a)$ and $F(a)$, respectively. Let $\mathbf{E} = \{S(a), F(a) \mid a \in \mathbf{VAct}\} \cup \mathbf{Act}$; in the terminology of [14], this is the set of *events* and I shall use e to range over it. The operational semantics is given in terms of a set of next-state relations $\overset{e}{\Rightarrow}$, one for each $e \in \mathbf{E}$. As explained at length in [14], the relations $\overset{e}{\Rightarrow}$ are defined over the set of states **S**, a superlanguage of **P** obtained by adding new prefixing operators a_S to the formation rules for **P**. I shall use s, s', s_1, s_2 to range over the set of states **S**. The relations $\overset{e}{\Rightarrow}$ are defined to be the least ones over **S** that satisfy the rules in Fig. 1. Comments on these rules may be found in §2.1 of [14].

The relation of *t-observational equivalence* $\approx_T$ is now defined as the largest symmetric relation on states which satisfies

$s_1 \approx_T s_2$ if and only if for every $e \in \mathbf{E}$, $s_1 \overset{e}{\Rightarrow} s_1'$ implies
(i) $e = \tau$ and $s_1' \approx_T s_2$, or
(ii) $s_2 \overset{e}{\Rightarrow} s_2'$ for some s_2' such that $s_1' \approx_T s_2'$.

Following Hennessy, I shall only be interested in $\approx_T$ as it applies to the language of processes **P**. The interested reader is referred to §2.1 of [14] for examples of equivalent and inequivalent processes with respect to $\approx_T$. Here, I shall only remark that t-observational equivalence is a noninterleaving equivalence, in whose theory the concurrent execution of actions is discriminated from their arbitrary interleaving by means of information about the structure of actions. For example, the processes $a.\mathbf{0} \parallel b.\mathbf{0}$ and $a.b.\mathbf{0} + b.a.\mathbf{0}$ are inequivalent with respect to $\approx_T$ because the former can start an a action and subsequently start a b action, whilst the latter cannot.

The equivalence $\approx_T$ is not a congruence over **P** for the usual reasons associated with the operators $+$ and $\lfloor$. For example, it is easy to see that $\mathbf{0} \approx_T \tau.\mathbf{0}$, but

$$\mathbf{0} \lfloor a.\mathbf{0} \approx_T \mathbf{0} \not\approx_T \tau.a.\mathbf{0} \approx_T \tau.\mathbf{0} \lfloor a.\mathbf{0}.$$

$$\frac{}{a.p \overset{S(a)}{\Rightarrow} a_S.p} \qquad \frac{}{a_S.p \overset{F(a)}{\Rightarrow} p} \qquad \frac{}{\mu.p \overset{\mu}{\Rightarrow} p}$$

$$\frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_1 + s_2 \overset{e}{\Rightarrow} s_1'} \qquad \frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_2 + s_1 \overset{e}{\Rightarrow} s_1'}$$

$$\frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_1 \| s_2 \overset{e}{\Rightarrow} s_1' \| s_2} \qquad \frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_2 \| s_1 \overset{e}{\Rightarrow} s_2 \| s_1'} \qquad \frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_1 \lceil s_2 \overset{e}{\Rightarrow} s_1' \| s_2}$$

$$\frac{s_1 \overset{a}{\Rightarrow} s_1',\ s_2 \overset{\bar{a}}{\Rightarrow} s_2'}{s_1 \| s_2 \overset{\tau}{\Rightarrow} s_1' \| s_2'} \qquad \frac{s_1 \overset{a}{\Rightarrow} s_1',\ s_2 \overset{\bar{a}}{\Rightarrow} s_2'}{s_1 \lceil s_2 \overset{\tau}{\Rightarrow} s_1' \| s_2'}$$

$$\frac{s_1 \overset{\tau}{\Rightarrow} s_1',\ s_1' \overset{e}{\Rightarrow} s_2}{s_1 \overset{e}{\Rightarrow} s_2} \qquad \frac{s_1 \overset{e}{\Rightarrow} s_1',\ s_1' \overset{\tau}{\Rightarrow} s_2}{s_1 \overset{e}{\Rightarrow} s_2}$$

FIG. 1. *Operational rules for* $\overset{e}{\Rightarrow}$.

One of the main results in [14] is a complete equational characterization of the largest congruence $\approx_T^C$ contained in $\approx_T$ over the set of processes. (See Theorem 2.1.2 in [14].) For ease of reference, Hennessy's equations for $\approx_T^C$ are collected in Fig. 2.

Unfortunately, however, the axiomatization presented in Fig. 2 is incorrect. This is due to the fact that axiom (B2), which plays a vital role in the reduction of terms to *simple forms* (see the proof of Proposition 2.2.3 in [14]), is unsound as the following example shows. (A similar example may be found on page 142 of [7].)

Example. Consider the terms $p \equiv (a.c.\mathbf{0} \lceil b.\mathbf{0}) \lceil \bar{b}.\bar{a}.\mathbf{0}$ and $q \equiv a.c.\mathbf{0} \lceil (b.\mathbf{0} \| \bar{b}.\bar{a}.\mathbf{0})$. I claim that $p \not\approx_T^C q$. In fact, using the rules in Fig. 1, it is easy to see that $b.\mathbf{0} \| \bar{b}.\bar{a}.\mathbf{0} \overset{\bar{a}}{\Rightarrow} \mathbf{0} \| \mathbf{0}$. This allows one to derive that $q \overset{c}{\Rightarrow} \mathbf{0} \| (\mathbf{0} \| \mathbf{0})$. On the other hand, p cannot initially perform a c action.

The problem in the equational characterization of Hennessy's auxiliary operator $\lceil$ derives from the fact that, although simpler than $\|$, $\lceil$ still captures two conceptually distinct features of parallel composition. One of them is the asynchronous behaviour due to one of the parallel components; the other is synchronization between processes. In their work on ACP, Bergstra and Klop have used two auxiliary operators, namely, *left-merge* $\lfloor\!\lfloor$ and *communication merge* $|$, to give a finite equational axiomatization of the parallel composition operation. Intuitively, the left-merge operation is used to capture the behaviour of parallel composition due to one of the parallel components and the communication merge is used to capture the behaviour deriving from synchronization. In the remainder of this section, I shall present an equational characterization of $\approx_T^C$ over $\mathbf{P}$ which will make a fundamental use of a noninterleaving variation on Bergstra and Klop's auxiliary operations[1]. All my attempts to find a sound and complete axiom system for $\mathbf{P}$ without the introduction of Bergstra and Klop's auxiliary operators have been to no avail.

[1] Bergstra and Klop's left-merge operation satisfies axiom (NLM2) in Fig. 4, whilst the left-merge operation I shall use in this section does not.

$$
\begin{array}{lrcl}
\text{A1} & (x+y)+z & = & x+(y+z) \\
\text{A2} & x+y & = & y+x \\
\text{A3} & x+x & = & x \\
\text{A4} & x+\mathbf{0} & = & x \\
\\
\text{B1} & (x+y)\lceil z & = & x\lceil z+y\lceil z \\
\text{B2} & (x\lceil y)\lceil z & = & x\lceil (y\|z) \\
\text{B3} & x\lceil \mathbf{0} & = & x \\
\text{B4} & \mathbf{0}\lceil x & = & \mathbf{0} \\
\\
\text{I1} & x+\tau.x & = & \tau.x \\
\text{I2} & \mu.\tau.x & = & \mu.x \\
\text{NI3} & x\lceil (y+\tau.z) & = & x\lceil (y+\tau.z)+x\lceil z \\
\\
\text{X1} & x\|y & = & x\lceil y+y\lceil x \\
\text{NX2} & \tau.x\lceil y & = & \tau.(x\|y) \\
\text{NX3} & x\lceil \tau.y & = & x\lceil y \\
\\
\text{C} & a.x_1\lceil ((\bar{a}.x_2\lceil y)+z) & = & a.x_1\lceil ((\bar{a}.x_2\lceil y)+z)+\tau.(x_1\|x_2\|y)
\end{array}
$$

FIG. 2. *Hennessy's equations for* $\approx_T^C$.

Let $\mathbf{P}_{\text{ext}}$ denote the language obtained by extending the grammar for $\mathbf{P}$ with the following formation rule:

if $p, q \in \mathbf{P}_{\text{ext}}$, then $p \mathbin{|\!\!\lfloor} q \in \mathbf{P}_{\text{ext}}$ and $p|q \in \mathbf{P}_{\text{ext}}$.

The set of states $\mathbf{S}_{\text{ext}}$ associated with the extended language $\mathbf{P}_{\text{ext}}$ is defined in exactly the same way as $\mathbf{S}$. The operational semantics for the language of extended states $\mathbf{S}_{\text{ext}}$ is obtained by adding the following rules for the new operators to those in Fig. 1:

$$\frac{s_1 \overset{e}{\Rightarrow} s_1'}{s_1 \mathbin{|\!\!\lfloor} s_2 \overset{e}{\Rightarrow} s_1'\|s_2} \qquad \frac{s_1 \overset{a}{\Rightarrow} s_1',\ s_2 \overset{\bar{a}}{\Rightarrow} s_2'}{s_1|s_2 \overset{\tau}{\Rightarrow} s_1'\|s_2'}.$$

The notions of t-observational equivalence and t-observational congruence can now be conservatively extended to the language $\mathbf{P}_{\text{ext}}$ and, as in Lemmas 2.1.1 and 2.2.1 in [14], the following results hold.

LEMMA 2.1. *For all* $p, q \in \mathbf{P}_{\text{ext}}$,

(1) $p \approx_T^C q$ *if and only if* $p + a.\mathbf{0} \approx_T q + a.\mathbf{0}$ *for some* $a \in \mathbf{VAct}$ *not occurring in* p *and* q;

(2) $p \approx_T q$ *if and only if* $p \approx_T^C q$ *or* $\tau.p \approx_T^C q$ *or* $p \approx_T^C \tau.q$.

A standard, useful corollary of the characterization given in statement (1) of the above lemma is that if $p \approx_T^C q$, then p and q must have matching τ-transitions. (See, e.g., [14, p. 1010].)

I shall now address the problem of giving a sound and complete axiomatization of t-observational congruence over the language $\mathbf{P}_{\text{ext}}$ and, hence, over its sublanguage $\mathbf{P}$. First of all, note that sound versions of equations (B1)–(B4) may be given by replacing Hennessy's $\lceil$ with the left-merge operator. In particular, the following variation on equation (B2) holds in the quotient algebra $\mathbf{P}_{\text{ext}}/\approx_T^C$:

$$(x \mathbin{|\!\!\lfloor} y) \mathbin{|\!\!\lfloor} z = x \mathbin{|\!\!\lfloor} (y\|z).$$

The key to the soundness of the above equation is the fact that the left-merge operation does not allow for synchronization between its operands. For example, the reader can easily adapt the aforementioned example showing the unsoundness of axiom (B2) to prove that a version of the above equation in terms of the communication merge is not valid in $\mathbf{P}_{\mathbf{ext}}/\approx_T^C$, i.e., that there are processes $p, q, r \in \mathbf{P}_{\mathbf{ext}}$ such that

$$(p \mid q) \mid r \not\approx_T^C p \mid (q \| r).$$

Synchronization between processes is described by the communication merge operator. In fact, left-merge and communication merge together allow one to describe equationally the behaviour of parallel composition and of Hennessy's $\upharpoonright$ operator. The relevant equations are

$$(x \upharpoonright y) = x \mathbin{\|\!\!\lfloor} y + x \mid y, \tag{1}$$

$$x \| y = x \mathbin{\|\!\!\lfloor} y + y \mathbin{\|\!\!\lfloor} x + x \mid y. \tag{2}$$

Equation (1) was given in [5] in a setting without internal actions, while equation (2) is the key to the finite axiomatizations of bisimulation congruences presented in many papers in the literature on ACP. (See, e.g., [6].) Note that, in the presence of left-merge and communication merge, Hennessy's merge operator is no longer necessary to axiomatize CCS parallel composition.

The communication merge operator satisfies, among other laws, the following version of axiom (C):

$$(a.x \mathbin{\|\!\!\lfloor} y) \mid (b.w \mathbin{\|\!\!\lfloor} z) = \begin{cases} \tau.(x \| y \| w \| z) & \text{if } a = \bar{b}, \\ \mathbf{0} & \text{otherwise,} \end{cases}$$

where I have taken the liberty of omitting a cumbersome use of parentheses because parallel composition is commutative and associative modulo $\approx_T^C$. An equation expressing a fundamental property of the communication merge operator in the theory of t-observational congruence is the following:

$$\tau.x \mid y = x \mid y.$$

This law was first presented in [6], where it was shown to be sound with respect to a graph model for Milner's observational congruence (or rooted τ-bisimulation equivalence, in Bergstra and Klop's terminology). It expresses the subtle interplay between internal τ-actions and synchronization in the theory of observational congruence-like relations.

The set $\mathcal{E}$ of equations which make up the axiomatization of t-observational congruence over $\mathbf{P}_{\mathbf{ext}}$ is given in Fig. 3. The main result of this paper may now be stated.

THEOREM 2.2. *For all* $p, q \in \mathbf{P}_{\mathbf{ext}}$, $p \approx_T^C q$ *if and only if* $\mathcal{E} \vdash p = q$.

I shall now sketch the steps involved in the proof of Theorem 2.2. The presentation will closely follow §2.2 in [14] and the interested reader is referred to that reference for many details.

The first step in the proof of Theorem 2.2 is to show that all the equations in $\mathcal{E}$ are indeed satisfied by $\approx_T^C$. This is the import of the following result, whose proof is straightforward but rather tedious.

PROPOSITION 2.3 (Soundness). *For all* $p, q \in \mathbf{P}_{\mathbf{ext}}$, $\mathcal{E} \vdash p = q$ *implies* $p \approx_T^C q$.

The proof of the completeness of the equations in $\mathcal{E}$ with respect to t-observational congruence follows the general outline of that of Theorem 2.1.2 in [14]. As usual, I shall rely on the existence of normal forms for processes. These are very similar to Hennessy's simple

$$
\begin{array}{lrcll}
\text{A1} & (x+y)+z & = & x+(y+z) & \\
\text{A2} & x+y & = & y+x & \\
\text{A3} & x+x & = & x & \\
\text{A4} & x+\mathbf{0} & = & x & \\
& & & & \\
\text{LM1} & (x+y)\lfloor\!\lfloor z & = & x\lfloor\!\lfloor z+y\lfloor\!\lfloor z & \\
\text{LM2} & (x\lfloor\!\lfloor y)\lfloor\!\lfloor z & = & x\lfloor\!\lfloor (y\|z) & \\
\text{LM3} & x\lfloor\!\lfloor \mathbf{0} & = & x & \\
\text{LM4} & \mathbf{0}\lfloor\!\lfloor x & = & \mathbf{0} & \\
& & & & \\
\text{I1} & x+\tau.x & = & \tau.x & \\
\text{I2} & \mu.\tau.x & = & \mu.x & \\
\text{ILM1} & x\lfloor\!\lfloor (y+\tau.z) & = & x\lfloor\!\lfloor (y+\tau.z)+x\lfloor\!\lfloor z & \\
\text{ILM2} & \tau.x\lfloor\!\lfloor y & = & \tau.(x\|y) & \\
\text{ILM3} & x\lfloor\!\lfloor \tau.y & = & x\lfloor\!\lfloor y & \\
& & & & \\
\text{CM1} & (x+y)\mid z & = & x\mid z+y\mid z & \\
\text{CM2} & x\mid y & = & y\mid x & \\
\text{CM3} & x\mid \mathbf{0} & = & \mathbf{0} & \\
\text{CM4} & (a.x\lfloor\!\lfloor y)\mid(b.w\lfloor\!\lfloor z) & = & \begin{cases} \tau.(x\|y\|w\|z) & \text{if } a=\bar{b}, \\ \mathbf{0} & \text{otherwise} \end{cases} & \\
\text{CM5} & \tau.x\mid y & = & x\mid y & \\
& & & & \\
\text{PAR} & x\|y & = & x\lfloor\!\lfloor y+y\lfloor\!\lfloor x+x\mid y & \\
\text{HM} & x\lceil y & = & x\lfloor\!\lfloor y+x\mid y &
\end{array}
$$

FIG. 3. *Complete equations for* $\approx_T^C$ *over* $\mathbf{P}_{\text{ext}}$.

forms. (See Definition 2.2.2 in [14].) As usual in the literature on process algebras, the notation $\sum\{p_i \mid i \in I\}$ is used as a shorthand for $p_{i_1}+\cdots+p_{i_n}$ where $I=\{i_1,\ldots,i_n\}$. If $I=\emptyset$, then $\sum\{p_i \mid i \in \emptyset\} \equiv \mathbf{0}$.

DEFINITION 2.4. *The set of* normal forms **NF** *is the least subset of* $\mathbf{P}_{\text{ext}}$ *such that*

$\sum\{a_i.p_i\lfloor\!\lfloor p_i' \mid i \in I\}+\sum\{\tau.q_j \mid j \in J\} \in \mathbf{NF}$ *if* I, J *are finite index sets and each* $p_i, p_i', q_j \in \mathbf{NF}$.

PROPOSITION 2.5 (Normalization). *For every process* $p \in \mathbf{P}_{\text{ext}}$, *there exists a normal form* $\hat{p} \in \mathbf{NF}$ *such that* $\mathcal{E} \vdash p = \hat{p}$.

Proof. The proof of this result is standard and many similar ones may be found in the literature. Detailed proofs for closely related languages may be found in, e.g., [7], [18]. Equations (A3), (I1), (I2), (ILM1), and (ILM3) are not needed in the proof. □

Following Hennessy, the proof of completeness of the set of equations $\mathcal{E}$ relies on establishing so-called "derivation lemmas." As in [14], I shall only be interested in derivations with respect to τ-actions and $S(a)$-actions.

LEMMA 2.6 (Derivation lemma). *Let* $p \in \mathbf{P}_{\text{ext}}$. *Then*

(1) $p \overset{\tau}{\Rightarrow} q$ *implies* $\mathcal{E} \vdash p = p+\tau.q$;

(2) $p \overset{S(a)}{\Rightarrow} a_S.p_1\|p_2$ *implies* $\mathcal{E} \vdash p = p+a.p_1\lfloor\!\lfloor p_2$.

Proof. By Proposition 2.5, it is sufficient to prove the above statements for normal forms. Assume then that p is of the form $\sum\{a_i.p_i\lfloor\!\lfloor p_i' \mid i \in I\}+\sum\{\tau.q_j \mid j \in J\}$.

(1) Then by induction on the length of the derivation $p \equiv \sum\{a_i.p_i \mathbin{|\!\lfloor} p'_i \mid i \in I\} + \sum\{\tau.q_j \mid j \in J\} \overset{\tau}{\Rightarrow} q$.

Base case: $q \equiv q_j$ for some $j \in J$. Then $\mathcal{E} \vdash p = p + \tau.q$ follows immediately by using equations (A1)–(A4).

Inductive step: $q_j \overset{\tau}{\Rightarrow} q$ for some $j \in J$. By the inductive hypothesis, it follows that $\mathcal{E} \vdash q_j = q_j + \tau.q$. By equations (I1) and (A1)–(A4), it is easy to derive that $\mathcal{E} \vdash \tau.q_j = \tau.q_j + \tau.q$, from which $\mathcal{E} \vdash p = p + \tau.q$ follows immediately.

(2) This statement is proven exactly as Corollary 2.2.5 in [14]. □

The key to the proof of the completeness theorem is an important decomposition result proven by Hennessy in [14] for the language **P**. The extension of Hennessy's result to the language $\mathbf{P_{ext}}$ is immediate, and, in fact, his proof carries over unchanged to this language.

PROPOSITION 2.7 (Hennessy). *For all $p, p', q, q' \in \mathbf{P_{ext}}$, $a_S.p \| p' \approx_T a_S.q \| q'$ implies $p \approx_T q$ and $p' \approx_T q'$.*

Proof. See the proof of Proposition 2.2.8 in [14] and those of the lemmas leading up to it. □

The above results are all that is needed in the proof of the completeness result to follow.

THEOREM 2.8 (Completeness). *For all $p, q \in \mathbf{P_{ext}}$, $p \approx_T^C q$ implies $\mathcal{E} \vdash p = q$.*

Proof. This is just a reworking of Hennessy's proof of Theorem 2.2.9 in [14] using the results given above. The interested reader will have no difficulty in filling in the details following Hennessy's proof. □

2.1. An axiomatization of observational congruence. As mentioned in the introduction, Hennessy's axiomatization of Milner's observational congruence in [14] was the first to use an auxiliary operator to give an equational characterization of parallel composition. For the sake of clarity and in order to support the discussion to follow, I shall now recapitulate the definitions of weak bisimulation equivalence and its associated congruence.

The relation of *weak bisimulation equivalence* $\approx$ is defined as the largest symmetric relation over **P** which satisfies

$p \approx q$ if and only if for every $\mu \in \mathbf{Act}$, $p \overset{\mu}{\Rightarrow} p'$ implies
(i) $\mu = \tau$ and $p' \approx_T q$, or
(ii) $q \overset{\mu}{\Rightarrow} q'$ for some q' such that $p' \approx_T q'$.

As usual, $\approx$ is not a congruence over **P**. The largest congruence relation contained in $\approx$ will be denoted by $\approx^C$ and will be referred to as *observational congruence*.

The key to the axiomatization of observational congruence presented in Theorem 1.3.4 of [14] is a version of Milner's *interleaving law* in terms of Hennessy's $\lceil$. This is the following conditional equation schema:

$$\text{(X2)}\ \frac{y = \sum\{\lambda_j.y_j \mid j \in J\}\ (J \text{ a finite index set})}{\mu.x \lceil y = \mu.(x \| y) + \sum\left\{\tau.(x \| y_j) \mid \mu = \bar{\lambda}_j\right\}}.$$

This equation schema plays a vital role in the reduction of process terms to the sumforms used by Hennessy and Milner in [16] and Hennessy in [14]. Unfortunately, however, it is *not* sound with respect to observational congruence as the following example shows.

Example. Consider the instance of the above equation obtained by taking $\mu \equiv a$, $x \equiv \mathbf{0}$, and $y \equiv \tau.\bar{a}.\mathbf{0}$. Then (X2) allows us to derive that $a.\mathbf{0} \lceil \tau.\bar{a}.\mathbf{0} = a.(\mathbf{0} \| \tau.\bar{a}.\mathbf{0})$. However, this equality does not hold in the quotient algebra $\mathbf{P}/{\approx^C}$ as $a.\mathbf{0} \lceil \tau.\bar{a}.\mathbf{0} \overset{\tau}{\Rightarrow} \mathbf{0} \| \mathbf{0}$, whilst obviously $a.(\mathbf{0} \| \tau.\bar{a}.\mathbf{0})$ has no comparable transition.

The reader will have noticed that, once again, the unsoundness of axiom (X2) derives from the fact that Hennessy's $\lceil$ allows for communication between its arguments. The above problem with the axiomatization presented in [14] can be solved by resorting to Bergstra and Klop's auxiliary operators. In fact, it is possible to extend conservatively observational congruence to the language $\mathbf{P}_{\mathrm{ext}}$ and give a sound and complete equational axiomatization of equality in the quotient algebra $\mathbf{P}_{\mathrm{ext}}/\approx^C$. In fact, all that is needed for this purpose is to add the following equations to those presented in Fig. 3:

$$a.x \mathbin{\lfloor\!\lfloor} y = a.(x \,\|\, y), \tag{3}$$

$$a.(x + \tau.y) = a.(x + \tau.y) + a.y. \tag{4}$$

Equation (3) is the one that essentially expresses the fact that observational congruence induces an interleaving semantics on processes. Together with the other equations for left-merge and communication merge it allows for the derivation of Milner's expansion theorem. (See, e.g., [6] for a detailed proof of this fact.) Equation (4) is Milner's "third τ-law." As it is well known (see, e.g., [14], p. 1010), this equation does not hold for $\approx^C_T$ because it strongly depends on the assumption of atomicity of action occurrences.

In the presence of (3), several of the equations in Fig. 3 are not necessary to give a complete equational characterization of observational congruence over the language $\mathbf{P}_{\mathrm{ext}}$. (Note, however, that those equations lead to more powerful axiomatic systems for what concerns provability of equivalences between *open terms* over sublanguages of $\mathbf{P}_{\mathrm{ext}}$. The interested reader is referred to [20], [13] for more on this issue.) Moreover, axiom (CM4) may be simplified to

$$a.x \mid b.y = \begin{cases} \tau.(x \,\|\, y) & \text{if } a = \bar{b}, \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

A complete set of axioms for $\approx^C$ is given in Fig. 4. Let $\mathcal{E}'$ denote the set of equations in Fig. 4.

THEOREM 2.9. *For all $p, q \in \mathbf{P}_{\mathrm{ext}}$, $p \approx^C q$ if and only if $\mathcal{E}' \vdash p = q$.*

Proof. This is just a reworking of many similar results in the literature (see, e.g., [16], [6], [19]), following the outline of the proof of Theorem 1.3.4 in [14, p. 1008]. □

3. Remarks on the operational semantics of Bergstra and Klop's auxiliary operators. The reader familiar with the literature on bisimulation semantics for CCS will have already noted that the operational semantics for the language $\mathbf{P}_{\mathrm{ext}}$ given in the previous section is slightly nonstandard. The rules in Fig. 1 and those for Bergstra and Klop's auxiliary operators define the so-called *weak transition relations* over the language $\mathbf{P}_{\mathrm{ext}}$ in one step, so to speak. This is in contrast to the developments in, e.g., [19], where the operational semantics of CCS is defined first in terms of single-step transition relations. These concrete transition relations are then used in the definition of the weak transition relations, which capture the intuition that τ-labeled transitions correspond to invisible events. For easy reference, the defining rules of the one-step transition relations, $\xrightarrow{e}$, for the language $\mathbf{P}_{\mathrm{ext}}$ are collected in Fig. 5. The associated transition relations that abstract from internal τ-transitions are then usually defined by

$$s \overset{e}{\Rightarrow} s' \Leftrightarrow \exists s_1, s_2 : s \xrightarrow{\tau}{}^{\star} s_1 \xrightarrow{e} s_2 \xrightarrow{\tau}{}^{\star} s',$$

where $\xrightarrow{\tau}{}^{\star}$ denotes the reflexive and transitive closure of the relation $\xrightarrow{\tau}$.

The process of abstraction from τ-labeled transitions is instead built in the definition of the transition relations $\overset{e}{\Rightarrow}$ by means of the rules

$$\frac{s_1 \overset{\tau}{\Rightarrow} s_1',\ s_1' \overset{e}{\Rightarrow} s_2}{s_1 \overset{e}{\Rightarrow} s_2} \qquad \frac{s_1 \overset{e}{\Rightarrow} s_1',\ s_1' \overset{\tau}{\Rightarrow} s_2}{s_1 \overset{e}{\Rightarrow} s_2}.$$

$$
\begin{array}{llcl}
\mathrm{A1} & (x+y)+z & = & x+(y+z) \\
\mathrm{A2} & x+y & = & y+x \\
\mathrm{A3} & x+x & = & x \\
\mathrm{A4} & x+\mathbf{0} & = & x \\
 & & & \\
\mathrm{LM1} & (x+y)\mathbin{\lfloor\!\lfloor} z & = & x\mathbin{\lfloor\!\lfloor} z+y\mathbin{\lfloor\!\lfloor} z \\
\mathrm{NLM2} & \mu.x\mathbin{\lfloor\!\lfloor} y & = & \mu.(x\|y) \\
\mathrm{LM4} & \mathbf{0}\mathbin{\lfloor\!\lfloor} x & = & \mathbf{0} \\
 & & & \\
\mathrm{I1} & x+\tau.x & = & \tau.x \\
\mathrm{I2} & \mu.\tau.x & = & \mu.x \\
\mathrm{I3} & a.(x+\tau.y) & = & a.(x+\tau.y)+a.y \\
 & & & \\
\mathrm{CM1} & (x+y)\mid z & = & x\mid z+y\mid z \\
\mathrm{CM2} & x\mid y & = & y\mid x \\
\mathrm{CM3} & x\mid\mathbf{0} & = & \mathbf{0} \\
\mathrm{NCM4} & a.x\mid b.y & = & \begin{cases} \tau.(x\|y) & \text{if } a=\bar{b}, \\ \mathbf{0} & \text{otherwise} \end{cases} \\
\mathrm{CM5} & \tau.x\mid y & = & x\mid y \\
 & & & \\
\mathrm{PAR} & x\|y & = & x\mathbin{\lfloor\!\lfloor} y+y\mathbin{\lfloor\!\lfloor} x+x\mid y \\
\mathrm{HM} & x\lceil y & = & x\mathbin{\lfloor\!\lfloor} y+x\mid y
\end{array}
$$

FIG. 4. *Complete equations for* $\approx^C$ *over* $\mathbf{P}_{\mathrm{ext}}$.

It is easy to see that, for processes in $\mathbf{P}_{\mathrm{ext}}$ not containing occurrences of the communication merge and of Hennessy's $\lceil$, the weak transition relations $\overset{e}{\Rightarrow}$ and $\overset{e}{\Rrightarrow}$ are in complete agreement, i.e., for all such s,

$$s \overset{e}{\Rightarrow} s' \Leftrightarrow s \overset{e}{\Rrightarrow} s'.$$

In particular, this implies that observational congruence and t-observational congruence over the sublanguage of $\mathbf{P}_{\mathrm{ext}}$ consisting of these terms can be defined using either of these two transition relations.

This agreement does, however, break down for terms having the communication merge operator or Hennessy's $\lceil$ as head operator. Consider, for example, the term $p \equiv \tau.a \mid \bar{a}.b.\mathbf{0}$. Then, using the defining rules for $\dot{\Rightarrow}$, it is easy to derive that $p \overset{b}{\Rightarrow} \mathbf{0}\|\mathbf{0}$. However, p has no outgoing transition according to $\dot{\Rrightarrow}$, as rule

$$\frac{s_1 \xrightarrow{a} s_1',\, s_2 \xrightarrow{\bar{a}} s_2'}{s_1 \mid s_2 \xrightarrow{\tau} s_1'\|s_2'}$$

is not applicable to it. This fact has disastrous consequences in the theory of congruences that, like observational congruence and t-observational congruence, satisfy axiom

$$(\mathrm{I1}) \quad x+\tau.x \;=\; \tau.x.$$

In fact, the communication merge operation and Hennessy's $\lceil$ would not preserve any such congruence, if their operational semantics were given in terms of $\Rrightarrow$.

$$\frac{}{a.p \xrightarrow{S(a)} a_S.p} \qquad \frac{}{a_S.p \xrightarrow{F(a)} p} \qquad \frac{}{\mu.p \xrightarrow{\mu} p}$$

$$\frac{s_1 \xrightarrow{e} s_1'}{s_1 + s_2 \xrightarrow{e} s_1'} \qquad \frac{s_1 \xrightarrow{e} s_1'}{s_2 + s_1 \xrightarrow{e} s_1'}$$

$$\frac{s_1 \xrightarrow{e} s_1'}{s_1 \| s_2 \xrightarrow{e} s_1' \| s_2} \qquad \frac{s_1 \xrightarrow{e} s_1'}{s_2 \| s_1 \xrightarrow{e} s_2 \| s_1'} \qquad \frac{s_1 \xrightarrow{e} s_1'}{s_1 \mathbin{\lfloor\!\lfloor} s_2 \xrightarrow{e} s_1' \| s_2} \qquad \frac{s_1 \xrightarrow{e} s_1'}{s_1 \lceil s_2 \xrightarrow{e} s_1' \| s_2}$$

$$\frac{s_1 \xrightarrow{a} s_1',\ s_2 \xrightarrow{\bar{a}} s_2'}{s_1 \| s_2 \xrightarrow{\tau} s_1' \| s_2'} \qquad \frac{s_1 \xrightarrow{a} s_1',\ s_2 \xrightarrow{\bar{a}} s_2'}{s_1 \mid s_2 \xrightarrow{\tau} s_1' \| s_2'} \qquad \frac{s_1 \xrightarrow{a} s_1',\ s_2 \xrightarrow{\bar{a}} s_2'}{s_1 \lceil s_2 \xrightarrow{\tau} s_1' \| s_2'}$$

FIG. 5. *Operational rules for* $\xrightarrow{e}$.

Example (in terms of observational congruence). Consider the terms $\tau.a.\mathbf{0}$ and $a.\mathbf{0}+\tau.a.\mathbf{0}$. As $\approx^C$ satisfies axiom (I1), one has that $\tau.a.\mathbf{0} \approx^C a.\mathbf{0} + \tau.a.\mathbf{0}$. However, if the semantics of the communication merge operator were given in terms of the rules in Fig. 5, it would be the case that

$$p \equiv \tau.a.\mathbf{0} \mid \bar{a}.b.\mathbf{0} \not\approx^C (a.\mathbf{0} + \tau.a.\mathbf{0}) \mid \bar{a}.b.\mathbf{0} \equiv q.$$

In fact, $q \overset{b}{\Rrightarrow} \mathbf{0}\|\mathbf{0}$, whilst, as remarked above, p has no outgoing transitions with respect to $\Rrightarrow$.

The outcome of this discussion is that a suitable operational semantics for the communication merge operator and Hennessy's $\lceil$ in the theory of congruences that, like those axiomatized in this paper, satisfy axiom (I1) can only be given by defining the weak transition relations in one step[2], as in Fig. 1. I believe that this observation was already implicit in the denotational semantics for ACP in terms of process graphs presented by Bergstra and Klop in their seminal paper [6], but, probably because of the denotational nature of the semantics presented in that reference, it seems to have gone unnoticed in several papers in the literature. (A notable exception is [9], where an interesting operational semantics for ACP along the lines of that in Fig. 1 has been presented.)

Indeed, contrary to what happens for the basic CCS combinators, the operational semantics of the auxiliary operators used in the axiomatization of parallel composition is highly sensitive to the kind of behavioural congruence one wants to impose on terms. A full discussion of this point would lead me too far from the main aim of this paper. Thus, I shall just end by giving a short "recipe book" for giving semantics to the auxiliary operators discussed in this paper in the setting of some of the best-known semantic theories for processes, with pointers to references where they are discussed in detail. These may be found in Fig. 6. I hope that they

[2] The point here is that one is really interested in the identifications induced by the chosen congruence over the basic language, e.g., CCS, used to write specifications of concurrent systems. Auxiliary operators are only added for axiomatization purposes and as an aid in algebraic manipulations of terms. Hence, one should like to add these operations *conservatively*, that is to say that their presence should not influence the equalities over the basic language. This, of course, requires that these new operations preserve the chosen behavioural congruence.

Behavioural Congruences	Suitable Transition Relations
Observational congruences satisfying (I1)	Define the semantics of the auxiliary operators by giving rules that give the weak transition relations in one step, as in Fig. 1.
Branching bisimulation congruence [12]	Define the semantics of the auxiliary operators by giving rules that give the one-step transition relation, as in Fig. 5. See, e.g., [3].
Testing and failures congruences	As remarked in [11], for these congruences the left-merge operator causes just as many problems as the other auxiliary operators. A possible solution, based on the use of the nondeterministic operators from CSP [17] in lieu of the CCS/ACP combination of sum and τ and on the possibility of giving a suitable one step transition relation for the modified language, may be found in [2].

FIG. 6. *A menagerie of suitable semantics for the auxiliary operators.*

will be a useful reference for researchers interested in complete axiomatizations of behavioural congruences.

Acknowledgments. I thank Matthew Hennessy for his encouragement to write this paper.

REFERENCES

[1] L. ACETO AND M. HENNESSY, *Towards action refinement in process algebras*, Inform. and Control, 103 (1993), pp. 204–269.

[2] L. ACETO AND A. INGÓLFSDÓTTIR, *A theory of testing for ACP*, in Proceedings CONCUR 91, Amsterdam, J. Baeten and J. Groote, eds., Lecture Notes in Comput. Sci. 527, Springer-Verlag, New York, 1991, pp. 78–95.

[3] J. BAETEN AND W. WEIJLAND, *Process algebra*, Cambridge Tracts Theoret. Comput. Sci. 18, Cambridge Univ. Press, London, 1990.

[4] J. BERGSTRA AND J. KLOP, *Fixed point semantics in process algebras*, Report IW 206, Mathematisch Centrum, Amsterdam, 1982.

[5] ———, *Process algebra for synchronous communication*, Inform. and Comput., 60 (1984), pp. 109–137.

[6] ———, *Algebra of communicating processes with abstraction*, Theoretical Comput. Sci., 37 (1985), pp. 77–121.

[7] I. CASTELLANI, *Bisimulations for Concurrency*, Ph.D. thesis, Report CST-51-88, Department of Computer Science, University of Edinburgh, Scotland, April 1988.

[8] I. CASTELLANI AND M. HENNESSY, *Distributed bisimulations*, J. Assoc. Comput. Mach., 36 (1989), pp. 887–911.

[9] R. V. GLABBEEK, *Bounded nondeterminism and the approximation induction principle in process algebra*, in Proceedings STACS 87, F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., Lecture Notes in Comput. Sci. 247, Springer-Verlag, New York, 1987, pp. 336–347.

[10] R. V. GLABBEEK AND F. VAANDRAGER, *Petri net models for algebraic theories of concurrency*, in Proceedings PARLE conference, Eindhoven, Vol. II (Parallel Languages), J. D. Bakker, A. Nijman, and P. Treleaven, eds., Lecture Notes in Comput. Sci. 259, Springer-Verlag, New York, 1987, pp. 224–242.

[11] ———, *Modular specification of process algebras*, Theoret. Comput. Sci., 113 (1993), pp. 293–348.

[12] R. V. GLABBEEK AND W. WEIJLAND, *Branching time and abstraction in bisimulation semantics* (*extended abstract*), in Information Processing 89, G. Ritter, ed., North-Holland, Amsterdam, 1989, pp. 613–618; full version available as Report CS-R9120, CWI, Amsterdam, 1991.

[13] J. GROOTE, *A new strategy for proving ω–completeness with applications in process algebra*, in Proceedings CONCUR 90, Amsterdam, J. Baeten and J. Klop, eds., Lecture Notes in Comput. Sci. 458, Springer-Verlag, New York, 1990, pp. 314–331.

[14] M. HENNESSY, *Axiomatising finite concurrent processes*, SIAM J. Comput., 17 (1988), pp. 997–1017.

[15] ———, *A proof system for weak ST-bisimulation over a finite process algebra*, Computer Science Report 6/91, University of Sussex, June 1991, submitted for publication.

[16] M. HENNESSY AND R. MILNER, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., 32 (1985), pp. 137–161.

[17] C. HOARE, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, NJ, 1985.

[18] A. KIEHN, *Distributed bisimulations for finite CCS*, Tech. report 7/89, University of Sussex, United Kingdom, 1989.

[19] R. MILNER, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, NJ, 1989.

[20] F. MOLLER, *Axioms for Concurrency*, Ph.D. thesis, Report CST-59-89, Department of Computer Science, University of Edinburgh, Scotland, 1989.

SIAM J. COMPUT.
Vol. 23, No. 4, pp. 864–894, August 1994

THE COMPLEXITY OF MULTITERMINAL CUTS*

E. DAHLHAUS†, D. S. JOHNSON‡, C. H. PAPADIMITRIOU§, P. D. SEYMOUR¶, AND M. YANNAKAKIS‡

Abstract. In the multiterminal cut problem one is given an edge-weighted graph and a subset of the vertices called terminals, and is asked for a minimum weight set of edges that separates each terminal from all the others. When the number k of terminals is two, this is simply the mincut, max-flow problem, and can be solved in polynomial time. It is shown that the problem becomes NP-hard as soon as $k = 3$, but can be solved in polynomial time for planar graphs for any fixed k. The planar problem is NP-hard, however, if k is not fixed. A simple approximation algorithm for arbitrary graphs that is guaranteed to come within a factor of $2 - 2/k$ of the optimal cut weight is also described.

Key words. NP-completeness, network flow, partitioning, planar graphs

AMS subject classifications. 68Q25, 90B10

1. Introduction. The *multiterminal cut* problem can be defined as follows. Given a graph $G = (V, E)$, a set $S = \{s_1, s_2, \dots, s_k\}$ of k specified vertices or *terminals*, and a positive weight $w(e)$ for each edge $e \in E$, find a minimum weight set of edges $E' \subseteq E$ such that the removal of E' from E disconnects each terminal from all the others.

When $k = 2$ this problem reduces to the famous "min-cut/max-flow" problem, a problem of central significance in the field of combinatorial optimization because of its many applications and the fact that it can be solved in polynomial time (e.g., see [7], [17], [18], [20]). The "k-terminal cut" problem for $k > 2$ has been a subject of discussion in the combinatorics community for years (closely-related variants were proposed as early as 1969 by Hu [17, p. 150]). A variety of applications have been suggested, most having to do with the minimization of communication costs in parallel computing systems. In [23], Stone points out how the problem of assigning program modules to processors can be formulated in this framework. Other applications involve partitioning files among the nodes of a network, assigning users to base computers in a multicomputer environment and partitioning the elements of a circuit into the subcircuits that will go on different chips. It is known that such problems can become NP-hard even for $k = 2$ if there is a constraint imposed on the *size* of the components into which the graph is cut [9], [10]. In this paper we ask whether the problem might be tractable without such a constraint (as it is for $k = 2$).

Our first results concern the planar case. The restriction to planar graphs, besides its basic graph-theoretic significance, has potential relevance in the circuit partitioning application.

THEOREM 1.

(a) *For $k = 3$, the planar multiterminal cut problem can be solved in time $O(n^3 \log n)$.*

(b) *For any fixed $k \geq 3$, the planar multiterminal cut problem is solvable in polynomial time.*

The algorithms of Part (b) are, unfortunately, exponential in k. (Specifically, they are $O((4k)^k n^{2k-1} \log n)$.) That such exponential behavior is likely to be unavoidable follows from the next result.

*Received by the editors January 27, 1992; accepted for publication (in revised form) May 18, 1993. A preliminary version of this paper appeared as "The complexity of multiway cuts" in *Proc. 24th Annual ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York (1992), pp. 241-251.

†University of Bonn, Romerstrasse 164, D-5300 Bonn 1, Germany, Current Address, Basser Department of Computer Science, University of Sidney, New South Wales 2006, Australia. Department of Computer Science.

‡AT&T Bell Laboratories, Murray Hill, New Jersey 07974 (dsj@research.att.com).

§Department of Computer Science and Engineering, University of California at San Diego, 0014, 9500 Gilman Drive, La Jolla, California 92093-0114.

¶Bell Communication Research, Morristown, New Jersey 07960.

THEOREM 2. *If k is not fixed, the multiterminal cut problem for planar graphs is* NP*-hard even if all edge weights are equal to* 1.

For the multiterminal cut problem in arbitrary graphs, NP-hardness sets in much earlier.

THEOREM 3. *The multiterminal cut problem for arbitrary graphs is* NP*-hard for all fixed $k \geq 3$ even if all edge weights are equal to* 1.

This theorem is proved using a "gadget" that has interesting properties on its own (as a counterexample to a conjecture about the possible submodularity of 3-terminal cut). The theorem's negative consequences are partially mitigated by technical lemmas that may yield substantial reductions in the sizes of instances encountered in practice.

Finally, we have the following two approximation results, one positive and one negative.

THEOREM 4. *There is an $O(knm \log(n^2/m))$ approximation algorithm for the multiterminal cut problem that for arbitrary graphs and arbitrary k is guaranteed to find cuts that are within $2(k-1)/k$ of optimal.*

THEOREM 5. *For any fixed $k \geq 3$, the k-terminal cut problem is* MAX SNP*-hard (and hence cannot have a polynomial time approximation scheme unless P =*NP [1], [21]).

The results presented here can be contrasted to those of [13], [16], [22], which concern what might be called the *k-Cut* problem. In this problem we are given G, k, and w as above (but not S) and are asked merely for a minimum weight set of edges E' whose removal separates the graph into at least k nonempty connected components. Although this problem is NP-hard for arbitrary k, it is solvable in polynomial time for each fixed $k > 2$, even for arbitrary graphs [13]. The running time is $O(n^{k^2/2-k+11/2})$. Thus the k-cut problem is significantly easier than the problem we study here. Note, however, that this may not hold true in the case of planar graphs. At present there is no known general method for exploiting planarity in the k-cut problem, in contrast to our results for planar multiterminal cut. Thus for fixed $k \geq 6$, our planar multiterminal cut algorithm currently provides the best method for solving the planar k-cut problem: Simply run our algorithm for all possible sets S of k terminals and take the least-weight solution found. A factor proportional to n^k is added to our running time, but the resulting time bound is still $O(n^{3k-1} \log n)$, which beats $O(n^{k^2/2-k+11/2})$ when $k \geq 6$. (For $k = 3$ and *unweighted* planar graphs, the $O(n^7)$ of the general k-cut result has been beaten more directly, first by an $O(n^2)$ algorithm in [16], and subsequently by an $O(n \log n)$ algorithm in [15]). Reference [22] concerns approximation results for the k-cut problem, showing that the bounds we obtain in Theorem 4 for multiterminal cut can be obtained for k-cut directly, without having to apply our multiterminal result to all possible sets of k terminals.

To avoid bibliographic confusion, we should mention that, with the exception of Theorem 5, the results in the current paper were first announced in 1983 in an unpublished but widely circulated extended abstract [4]. The abstract has since been widely cited, both in the above-mentioned work on k-Cut and in follow-up work on the multiterminal cut problem itself. In [2], Chopra and Rao observe, as we failed to do in our original abstract, that for trees and 2-trees, the general k-Terminal Cut problem can be solved in linear time by a straightforward dynamic programming algorithm. (This can be generalized to graphs of bounded tree-width for any fixed bound, by standard techniques.) The facets of the multiterminal cut polyhedron are studied in [2], [3]. An interesting generalization of the multiterminal cut problem, about which we shall have more to say in our concluding section, is studied in [5], [6]. The 1983 abstract did not contain our proofs; these are presented here for the first time. (The 1983 abstract also used the less-descriptive term *multiway* cut for what we now call a multiterminal cut. The new terminology was introduced in [3] and we adopt it here for added clarity.)

The paper is organized as follows. In §2 we cover the positive results for the planar case (Theorem 1a and 1b). The corresponding negative result for the planar case (Theorem 2) is covered in §3. §4 covers our results for general graphs (Theorems 3, 4 and 5 and associated technical lemmas). A concluding §5 discusses additional variants and generalizations of multiterminal cut to which our techniques can apply and points out some of the remaining open problems in the area.

2. Algorithms for the planar case. Our main result for planar graphs (Theorem 1) says that for all fixed k, the multiterminal cut problem is solvable in polynomial time. This is in contrast to Theorem 3 that says that for arbitrary graphs the problem is NP-hard for any fixed $k \geq 3$. The key advantage we gain from planarity lies in the existence of a planar dual to our given graph G. We will assume without loss of generality that our graph $G = (V, E)$ is connected and that we have fixed an embedding of it on the plane. We will use a superscript D to denote a dual object. Thus G^D is the dual graph of G. If F is a subset of the edges of G, F^D is the corresponding set of edges of G^D. (Note: F^D is *not* the dual of the subgraph (V, F) of G.)

We start with Theorem 1a and the case of $k = 3$ and then show how our proof techniques can be generalized to cover the case of general fixed k (Theorem 1b).

2.1. Planar 3-terminal cuts. A key concept in all that follows is the idea of an *isolating cut.*

DEFINITION. *For a given terminal s_i, an* isolating cut *for s_i is any set of edges that cuts all paths between s_i and all the other terminals.*

Note that a minimum weight isolating cut for s_i can be constructed by merging all the terminals other than s_i into a special vertex s_0, and then finding a minimum $s_i - s_0$ cut in the resulting graph by a standard 2-terminal minimum cut algorithm. Note also that any k-terminal cut induces isolating cuts for each of the k terminals. An optimal k-terminal cut need not induce optimal isolating cuts however, because of the savings that can be obtained when the induced isolating cuts share edges. As we shall see, when G is planar, the sharing of edges has a convenient interpretation in terms of paths in the dual graph G^D.

For the purpose of introducing some terminology, let us for simplicity first look at the dual in the case when there is no sharing of edges. From now on in this section, we shall assume $k = 3$. Figure 1 shows a graph G with a 3-terminal cut C, together with the duals G^D and C^D of each. The thicker edges in the figure are those of C and C^D, respectively. Note that the edges of C^D partition the geometric embedding of G^D into three regions. (In the case of Fig. 1, two of these regions are single faces of G^D, but the other is the union of several faces.) Let us say that a vertex of G is *in* a given region if the face of G^D to which the vertex corresponds is part of that region. Then observe that in the figure, each of the terminals of G is in a separate region of C^D. This is clearly a general property: C is a 3-terminal cut of a graph G if and only if the terminals s_1, s_2, s_3 are in different regions of C^D. (The boundary of the region containing x_i is the dual of an isolating cut for x_i.) If C is an *optimal* 3-terminal cut, C^D has exactly three regions, each one containing a distinct terminal. Furthermore, removing any edge from C^D must merge two regions, as otherwise the corresponding edge of C is not needed in the cut.

For a general instance of the 3-terminal cut problem, there are two topologically distinct possibilities for an optimal cut C^D. See Fig. 2, where $\{i, j, k\} = \{1, 2, 3\}$.

Cut type I. C^D consists of two edge-disjoint cycles. (See Fig. 2a,b.) Note that the cycles may have one vertex in common and/or one cycle may lie inside the other, as in Fig. 2b. They cannot have more than one vertex in common, however, as this would imply that C^D had more than three regions.

Cut type II. Each pair of regions of C^D shares an edge. (See Fig. 2c.)

We shall now describe how to find an optimal 3-terminal cut. We provide procedures that

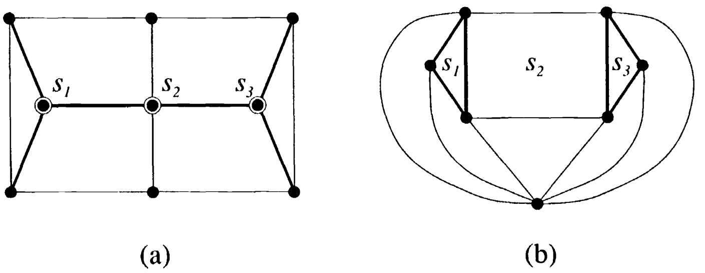


FIG. 1. *A planar 3-terminal cut* (a) *and its dual* (b).

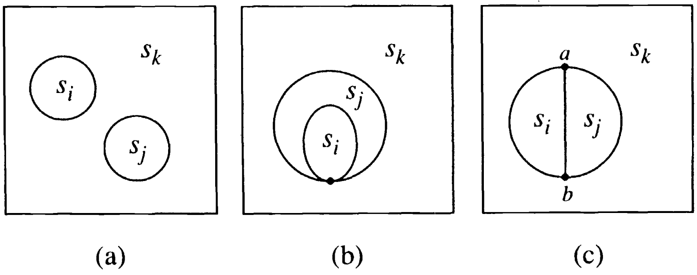


FIG. 2. *Types of 3-terminal cuts: type I* (a) *and* (b), *type II* (c).

work for each type of cut. Each procedure either returns the best cut of the corresponding type or else reports (correctly) that any optimal cut is of the other type.

Our procedure for Type I cuts is straightforward. We simply compute the three minimum weight isolating cuts for s_1, s_2, and s_3, respectively. Note that a minimum weight Type I cut must have weight at least as large as the sum of the weights of the two smallest of these three isolating cuts. If the two smallest are edge-disjoint, then their union is optimal among all 3-terminal cuts of Type I. If the two smallest are not edge-disjoint, then their union is a 3-terminal cut that has strictly smaller weight (because all edge weights are by assumption positive). Consequently, the best 3-terminal cut is not of Type I.

Our procedure for Type II cuts is significantly more complicated. Suppose we have an optimal 3-terminal cut that is of Type II. Look again at Fig. 2c. The cycle that bounds each region corresponds to an isolating cut for the terminal contained in that region, but these isolating cuts are not necessarily optimal, as they overlap. Consider the two vertices that are of degree 3 in C^D and are labeled a and b in the figure. The following lemma allows us to fix one of the three paths connecting a and b in G^D.

LEMMA 2.1. *Suppose that the dual of an optimal* 3*-terminal cut C is of Type II and a and b are the two vertices of degree* 3 *in C^D. Let P be any shortest path from a to b in G^D. Then there is an optimal* 3*-terminal cut C_0 that is of Type II, has a and b as its two vertices of degree* 3 *in C_0^D, and such that P is one of the three paths that join a to b in C_0^D.*

Proof. Consider all optimal 3-terminal cuts C_t such that C_t^D is of Type II with a and b as specified. Among these cuts, pick C_0 to be the cut C_t for which C_t^D contains the longest possible initial segment of P starting at a. We will show that C_0^D contains all of P.

Assume that P is *not* one of the three paths connecting a to b in C_0^D. As we traverse P from a to b, let x be the first vertex of P such that the edge leaving x in P is not in C_0^D. Let y be the first vertex of P after x that is in C_0^D. Note that it is possible that $x = a$ and/or $y = b$, or that x and y are consecutive vertices of P but the edge $\{x, y\}$ is not in C_0^D. Let Q_1, Q_2, Q_3

be the three $a-b$ paths in C_0^D. The path P initially follows one of these paths, say Q_1 without loss of generality, leaves it at vertex x, and then hits a path again at vertex y. We distinguish two cases depending on whether y is on the same path Q_1 as x (Fig. 3a) or on a different path, say without loss of generality Q_2 (Fig. 3b). In both cases the portion of P between x and y lies entirely in one region of C_0^D (by planarity), and partitions that region into two subregions. One of these two subregions contains the terminal that was contained in the original region, and the other contains no terminals at all.

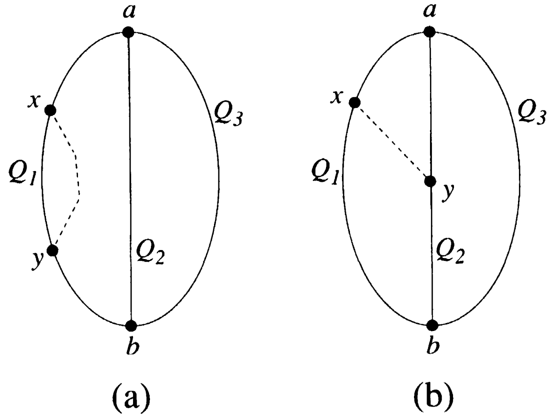


FIG. 3. *Possibilities for the* $P[x, y]$ *in the proof of Lemma* 2.1.

Case 1. Vertices x and y are both on Q_1. (This includes the cases when $x = a$ or $y = b$.) Let us denote the portion of a path Q between two of its vertices u and v as $Q[u, v]$. Let us assume without loss of generality that $P[x, y]$ lies in the region of C_0^D bounded by Q_1 and Q_2, as illustrated in Fig. 3a.

Subcase 1.1. The subregion bounded by $Q_1[x, y]$ and $P[x, y]$ contains no terminal.

Consider the path from a to b that follows P (and Q_1) from a to x, follows Q_1 from x to y, and then follows P from y to b. It must be at least as long as P because P is a shortest path from a to b. Therefore $w(Q_1[x, y]) \geq w(P[x, y])$, where if F is a set of edges, we take $w(F)$ to be $\Sigma_{e\in F} w(e)$. If we modify C_0^D so that $Q_1[x, y]$ is replaced by $P[x, y]$, we will obtain a path Q' that is no longer than Q_1, agrees with a longer initial segment of P and remains disjoint from Q_2 and Q_3 (except at their endpoints). Because the subregion bounded by $Q_1[x, y]$ and $P[x, y]$ contained no terminals, the union of Q' with Q_2 and Q_3 will still constitute the dual of a 3-terminal cut. This thus contradicts our definition of C_0, and so the subcase cannot apply.

Subcase 1.2. The subregion bounded by $Q_1[x, y]$ and $P[x, y]$ contains the terminal that lies in the region of C_0^D bounded by Q_1 and Q_2 (and hence the region bounded by Q_2 and the path consisting of $Q_1[a, x]$, $P[x, y]$, and $Q_1[y, b]$ contains no terminal).

In this case, we propose to modify C_0^D by deleting the path Q_2 and adding the path $P[x, y]$. Given the location of the terminal that was in the region of C_0^D bounded by Q_1 and Q_2, this will still be the dual of a 3-terminal cut. Thus all we must do now is argue that such a cut would violate the definition of C_0. First, observe that because P is a shortest path between its endpoints, all subpaths of P must themselves be shortest paths between their endpoints. In particular, we must have $w(P[x, y]) \leq w(Q_1[y, b]) + w(Q_2) + w(Q_1[a, x])$. If either $x \neq a$ or $y \neq b$, we would then have $w(P[x, y]) < w(Q_2)$, because all edge weights are positive by definition. Thus our modified 3-terminal cut would be strictly lighter than C_0, contradicting its definition. Therefore we must have $x = a$, $y = b$, and the current 3-terminal cut contains no edges from P. Note, however, that by our choice of P we have $w(P) \leq w(Q_2)$, and so we can replace Q_2 by P and obtain a new 3-terminal cut whose weight is at least as small. The new cut is hence also optimal and contradicts our assumption that no optimal cut could contain a longer subpath of P (starting from a) than does the cut containing Q_2. Thus this subcase cannot hold either, and Case 1 is ruled out.

Case 2. Vertex x is on $Q_1, x \neq a$, and vertex y is on Q_2, $y \neq b$.

Subcase 2.1. The region bounded by $Q_1[a, x]$, $P[x, y]$, and $Q_2[a, y]$ contains no terminal.

In this case, replacing $Q_2[a, y]$ by $P[x, y]$ in C_0^D will yield the dual of a 3-terminal cut. We argue that this new 3-terminal cut must have strictly smaller weight than C_0, contradicting the definition of C_0. Consider the path from a to b consisting of $Q_2[a, y]$ and $P[y, b]$. It must be at least as long as P, so we must have $w(Q_2[a, y]) \geq w(P[a, y])$. Consequently, because $a \neq x$, we must have $w(Q_2[a, y]) > w(P[x, y])$, and the new cut indeed has smaller weight.

Subcase 2.2. The region bounded by $Q_1[x, b]$, $P[x, y]$, and $Q_2[y, b]$ contains no terminal.

In this case we replace $Q_2[y, b]$ by $P[x, y]$ and obtain a contradiction analogous to the one of Subcase 2.1.

Thus Case 2 as well as Case 1 is ruled out. Consequently, P is contained in C_0, and the optimal 3-terminal cut called for by Lemma 2.1 exists. □

In light of Lemma 2.1, our procedure for Type II cuts can work by repeatedly calling a subroutine, once for each potential pair a, b of degree-3 vertices in C^D. The subroutine either constructs a minimum weight 3-terminal cut C that is of Type II and has a and b as the two degree-3 vertices in C^D, or reports (correctly) that no minimum weight 3-terminal cut has that form. The subroutine proceeds as follows.

First, construct a shortest path P between a and b in G^D. By Lemma 2.1 we may assume that P is contained in C^D. Delete the edges in G corresponding to the edges of P, obtaining a new graph H. In the embedding of this new graph induced by our original embedding of G, all the regions of G corresponding to vertices on P in G^D are merged into a single region. This corresponds in H^D to coalescing all the vertices along the path P from a to b into a single vertex v_P. This coalescence turns C^D from a Type II cut into a Type I cut like the one in Fig. 2b in which the two edge-disjoint cycles share a common vertex, in this case v_P. (The two cycles need not however be nested as they are in the figure; they can have disjoint interiors.)

We can now apply our previously described procedure for Type I cuts to H, obtaining a Type I cut C_H, or a report that the best 3-terminal cut for H is *not* of Type I. In the latter case, an optimal 3- terminal cut for G could not have been of Type II with the pair a, b as its degree-3 vertices, and we report this fact. In the former case, the cut C_H will, when augmented with the edges of G corresponding to the edges of P in G^D, be a 3-terminal cut for G. It cannot be an optimal cut, however, unless C_H^D has the desired form of two edge-disjoint cycles with v_P as a single common vertex. Otherwise the edges corresponding to P can be deleted and a valid (and lighter) 3-terminal cut for G will remain. Thus if C_H^D does not have the desired form, we once again report that no optimal 3-terminal cut for G is of Type II with a, b as its two degree-3 vertices.

Our overall algorithm for finding an optimal 3-terminal cut can thus proceed as follows.

Procedure 3-terminal.

1. Perform the Type I procedure on G.
   If a valid Type I cut is found, put it on the list of potential optima.
2. Construct the dual graph G^D and perform an all-pairs shortest path computation for G^D. For each pair a, b of vertices in G^D, do the following:

2.1. Let P be the shortest path in G^D between a and b as constructed in step 2 and let H be the graph obtained from G by deleting the edges corresponding to edges of P.
2.2. Perform the Type I procedure on H.
2.3. Let v_P be the coalesced vertex in H^D corresponding to the path P. If a valid Type I cut C_H for H is found and has a dual consisting of two edge-disjoint cycles having v_P as their unique common vertex, do the following:
 2.3.1 Let C_G be the 3-terminal cut for G consisting of C_H together with the edges of G corresponding to the edges of P in G^D.
 2.3.2 Add C_G to the list of potential optima.
3. Output the lightest 3-terminal cut on the list of potential optima.

THEOREM 1a. *Given a planar graph G with specified terminals s_1, s_2, and s_3, Procedure* 3-*terminal outputs an optimal* 3-*terminal cut and can be implemented to run in time $O(n^3 \log n)$, where n is the number of vertices in G.*

Proof. The fact that Procedure 3-terminal outputs an optimal cut follows from the above discussion. To analyze the running time, note that the bulk of the time is spent in the all-pairs shortest path computation of Step 2 and the isolating cut computations needed for each of the $\binom{n}{2}$ invocations of the Type I procedure in Step 2.2. The all-pairs shortest path computations takes place in a planar graph and so can be implemented to run in time $O(n^2)$ using the techniques of [8]. The isolating cut computations reduce as noted to 2-terminal minimum cut computations and so can be performed using standard 2-terminal cut algorithms.

For planar graphs, such algorithms run in time $O(n \log n)$, again using techniques from [8]. Unfortunately, if we use the techniques we originally described for computing isolating cuts, the graphs to which the 2-terminal cut algorithm is applied will not necessarily be planar. (Recall that our original proposal was to apply the 2-terminal cut algorithm to graphs constructed from G by coalescing pairs of terminals and note that those pairs need not be adjacent in G.) Thus without a further idea, we might be forced to use a general algorithm, and the running time would grow to $O(n^2 \log n)$. (This can be obtained for instance by using the $O(nm \log(n^2/m))$ 2-terminal cut algorithm of [12] and taking advantage of the fact that although our graphs need not remain planar, they do remain sparse.) This would force our overall running time up to $O(n^4 \log n)$. Fortunately, we can get around this obstacle as follows.

Recall that our goal in the Type I procedure is to find the two lightest among the isolating cuts for s_1, s_2, and s_3. For $i \neq j \in \{1, 2, 3\}$, let $c(i)$ denote the weight of a minimum isolating cut for s_i and $c(i, j)$ denote the weight of a minimum (2-terminal) cut separating s_i from s_j. Clearly, both $c(i), c(j) \geq c(i, j)$. Now note that in any cut separating s_i from s_j, the third terminal will be disconnected from at least one of s_i, s_j, and therefore the cut must isolate either s_i or s_j. Thus either $c(i) \leq c(i, j)$ or $c(j) \leq c(i, j)$. Combining this with the previous inequality, it follows that $c(i, j) = \min\{c(i), c(j)\}$. We compute the best two isolating cuts as follows.

Compute a minimum 2-terminal cut $C_{1,2}$ separating s_1 from s_2 in G. This will be an isolating cut for one of the two terminals, say s_1 without loss of generality. At this point we have $c(1, 2) = c(1) \leq c(2)$. Now compute a minimum 2-terminal cut $C_{2,3}$ separating s_2 (the nonisolated terminal) from s_3 in G. If this second cut isolates s_2 we have $c(2, 3) = c(2) \leq c(3)$; if it isolates s_3 we have $c(2, 3) = c(3) \leq c(2)$. In either case $C_{1,2}$ and $C_{2,3}$ are the two lightest isolating cuts as desired, and both were computed in the original planar graph G.

Step 2.2 thus involves $2\binom{n}{2} \approx n^2$ planar 2-terminal cut computations, for an overall time of $O(n^3 \log n)$. Because this is the dominant component of the running time, it also provides a bound on the overall running time of Procedure 3-terminal, which thus obeys the claimed running time bound. □

2.2. Planar multiterminal cuts. In this section we turn to the case of k-terminal cuts where $k > 3$. The algorithm we present will work for all $k \geq 3$ and will have a running time that, although exponential in k, is polynomial whenever k is fixed. It can be viewed as a (major) generalization of the algorithm of the previous section for the $k = 3$ case. For our discussion here, it will be convenient to assume that no two subsets of edges have the same total weight. (We can make sure that the assumption is satisfied in various ways. For instance, if Δ is the weight of the lightest edge and the edges are ordered $e_1, e_2, \ldots, e_m$, we could use the revised edge weights $w'(e_i) = w(e_i) + \Delta/2^i$.) The key consequence of the assumption is that optimal cuts, shortest paths, etc. are unique, so that we can refer to *the* optimal cut, etc. A less desirable consequence is that the cost of doing additions and comparisons of edge weights may go up by a factor of n, given the large number of bits needed to represent them, but given that our main goal here is to show that running times are $O(n^{ck})$ for some c, a factor of n will not make a significant difference.

In the $k = 3$ case, we observed that the dual C^D of the optimal cut was a subgraph of G^D that partitioned the embedding of G^D into three regions, each containing a distinct terminal. We then reduced the problem to the computation of 2-terminal cuts and shortest paths by first guessing (i.e., trying all possibilities for) some information about C^D. In particular, we guessed the *topology* (whether the cut consisted of two edge-disjoint cycles or not) and (in the latter case) the identity of the two degree-3 nodes a and b in C^D.

For general $k \geq 3$, we follow the same approach. Assume as before that we have previously decided on some fixed embedding of G. Suppose C is the optimal k-terminal cut, and once again let C^D be the dual of C viewed as a subgraph of a predetermined planar embedding of G^D. Then C^D must partition the embedding of G^D into precisely k regions, each containing a distinct terminal. Our notion of a *topology* for C^D is derived as follows. Consider the connected components of C^D and call such a component *complex* if it contains more than one vertex that has degree three or more in C^D. Let $C_1^D, C_2^D, \ldots, C_q^D$ be an enumeration of the complex components of C^D, and for each i, $1 \leq i \leq q$, let N_i be the set of vertices with degree three or more in C_i^D. (Note that we must have $q \leq k - 1$, because every connected component of C^D must enclose at least one terminal, and the infinite region of C^D contains one terminal.) Let $N = \cup_{i=1}^q N_i$. The *topology* of C^D is simply the (unordered) partition of N given by the sets $N_1, N_2, \ldots, N_q$. See Fig. 4 for an example of a C^D and its topology, consisting of two sets N_1 and N_2. In the figure the vertices in N are highlighted. Note that the degree-6 vertex v_{16} does not participate in the topology because its connected component contains only one vertex of degree three or greater.

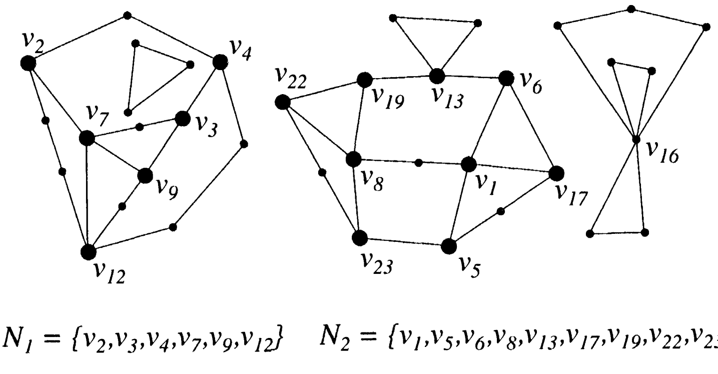

v2
v4
v7
v3
v9
v12
v22
v19
v13
v6
v8
v1
v17
v23
v5
v16
N1 = {v2,v3,v4,v7,v9,v12}
N2 = {v1,v5,v6,v8,v13,v17,v19,v22,v23}

FIG. 4. *The dual C^D of an optimal cut C and its topology* $\{N_1, N_2\}$.

LEMMA 2.2. *If G is a connected planar graph with n vertices, let C be an optimal k-terminal cut for G, and let C^D be the planar dual of C, viewed as a subgraph of G^D. Then the number of distinct possibilities for the topology of C^D is $O((2n)^{2k-4})$.*

Proof. Let an *arc* of C^D be any maximal path, all of whose interior vertices have degree two in C^D. Note that a cycle of C^D may be an arc, so long as the cycle contains at most one vertex of degree three or greater. Consider the graph whose vertex set is N and whose edge set is the arcs of C^D with both endpoints in N. This graph is planar, and has q connected components. Let m be the number of arcs it contains and $k' \leq k$, the number of regions. By Euler's formula, we have $|N| - m + k \geq q + 1$. Because all vertices of this graph have degree three or more by definition, we have $m \geq 3|N|/2$. Thus $|N| \leq 2(k - q - 1)$.

Now let V^D denote the set of vertices of G^D and consider the set of sequences $x_1, x_2, \ldots, x_{2k-4}$, with each x_i being either a member of V^D or the special symbol $|$. Such a sequence corresponds to a collection of subsets $S_1, S_2, \ldots$ of V^D in a natural way. Let us pretend our sequence is augmented by placing one copy of the special symbol $|$ at the beginning and one at the end. Then S_i is simply the set of vertices that occur in between the ith and $i+$ 1st occurrences of $|$ in the sequence. Note that every topology can be represented in this way, because all we need is $|N| + q - 1$ symbols, where N and q are as above, and $|N| + q - 1 \leq 2k - q - 3 \leq 2k - 4$ by the remark at the end of the preceding paragraph. Thus the total number of topologies is bounded by the total number of such sequences, which is $(|V^D| + 1)^{2k-4}$. (Note that this is a gross overcount on the topologies, because a given topology will be represented many times, and most sequences will not represent topologies at all. It probably represents the correct order of magnitude, however; for $k = 3$ it is $O(|V^D|^2)$, and we did have to consider $\Theta(|V^D|^2)$ topologies in the $k = 3$ case.)

To complete the proof of the lemma, we must bound $|V^D|$ in terms of n, the number of vertices in the original graph G. But note that each vertex in V^D corresponds to a region in the embedding of G, and (again by Euler's formula) the number of such regions is at most $2n - 4$. Thus the total number of topologies is $O((2n)^{2k-4})$ as claimed. □

Our algorithm for the general k-terminal cut problem will work by considering in turn each of the possibilities for the topology of the optimal cut. For each we will invoke a subroutine analogous to those used in our 3-terminal cut algorithm. The subroutine will either output the shortest cut that has the given topology, or else report (correctly) that the optimal cut cannot have the given topology. To specify the subroutine, we will need to know some structural results relating the optimum cut and its topology.

So suppose we are given a topology $N_1, N_2, \ldots, N_q$. If this is the optimal topology, then C^D contains connected components $C_1^D, C_2^D, \ldots, C_{q'}^D, q' \geq q$, where for $1 \leq i \leq q$, N_i is the set of vertices with degree three or more in C_i^D, and for $q < i \leq q'$, C_i^D contains at most one vertex with degree three or more. The analogue of Lemma 2.1 for this general k case is that for each C_i^D, $i \leq q$, we can efficiently identify a subtree T_i^D of C_i^D that spans all the vertices of N_i. This was the case in Lemma 2.1, where for $N_1 = \{a, b\}$ (Fig. 2c), we identified a path between the two degree-3 vertices a and b by doing a shortest path computation. For the general case, we shall need both shortest paths and minimum spanning trees.

For a given topology, the trees T_i^D, $i \leq q$, are constructed as follows. We treat the sets N_i, $i \leq q$, in turn. (Order is not important.) Given N_i, we construct an auxiliary weighted complete graph H_i with N_i as its vertex set and with the weight of the edge linking u and v being the length of the shortest path in G^D between u and v. Compute the minimum spanning tree $T[H_i]$ of H_i, and let T_i^D be the subgraph of G^D formed by replacing each edge in $T[H_i]$ by the corresponding shortest path in G^D. We shall call T_i^D the *minimum spanning tree* of N_i (although note that it may not even be a tree if C^D does not have the given topology). Figure 5 portrays the C^D of Fig. 4 with the trees T_i^D (chosen according to the correct topology) highlighted. (For future reference, the figure also indicates which terminal is contained in which region of C^D.) The next lemma establishes the key properties satisfied by T_i^D when C^D has the given topology.

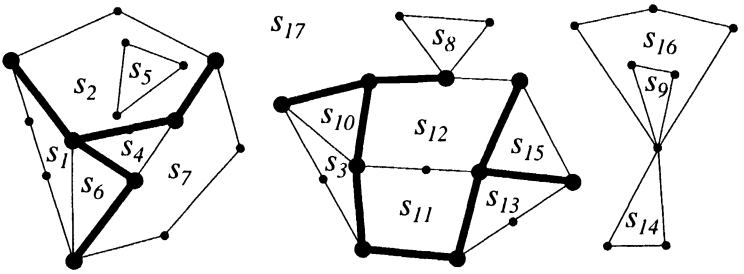


FIG. 5. *The dual C^D with the trees T_i^D highlighted and the locations of terminals indicated.*

LEMMA 2.3. *Let C be the optimal k-terminal cut and C^D be its planar dual, viewed as a subgraph of G^D. Let C_i^D, $1 \le i \le q$, be the complex connected components of C^D, and let N_i be the set of vertices with degree three or greater in C_i^D, $1 \le i \le q$. Then* (a) *each C_i^D, $1 \le i \le q$, contains the minimum spanning tree T_i^D for N_i, and* (b) *no two of the paths in T_i^D corresponding to edges of $T[H_i]$ intersect except at a common endpoint (and hence T_i^D is indeed a tree).*

Proof. Let us assume that the edges of $T[H_i]$ are labeled $F_1, F_2, \ldots, F_{|N_i|-1}$ in order of increasing weight. Thus if we let $T^j[H_i]$, $0 \le j < m$, be the graph consisting of the first j edges of $T[H_i]$, then for $0 \le j < m$, F_{j+1} is the shortest edge joining two different connected components (subtrees) of $T^j[H_i]$. Suppose $1 \le j \le m$, let $F_j = \{u, v\}$ and let S and T be the subtrees of $T^{j-1}[H_i]$ containing u and v, respectively. Let P be the shortest path in G^D between u and v. Because of our assumption that no two sets of edges have the same aggregate weight, we can prove the desired properties of P directly, without the induction that was needed in the proof of Lemma 2.1. We shall prove three claims, from which the current lemma follows. Claims 2 and 3 correspond to parts (a) and (b) of the lemma, respectively. Claim 1 is used in the proof of each of the latter two.

Claim 1. P contains no vertex of N_i other than u and v.

Note that if P contained such a vertex w, then either w is not in S or w is not in T. If w is not in S, then the edge $\{u, w\}$ in H_i connects two different subtrees and has length at most $P[u, w]$, which is strictly less than the length of P. This contradicts our assumption about the ordering of edges in $T[H_i]$. A similar argument applies to the path $P[w, v]$, if w is not in T. This gets us part of the way toward proving that no two paths of T_i^D intersect and will also be useful in Case 3 below.

Claim 2. P is entirely contained in C_i^D.

Suppose P is not entirely contained in C_i^D. As we traverse P from u to v, let x be the first vertex such that the edge leaving x is not included in C_i^D. Let y be the first vertex after x that is again included in C_i^D. As in the 3-terminal case, it is possible that $x = u$ and/or $y = v$, or that x and y are consecutive in P but $\{x, y\}$ is not in C_i^D. It is also possible that some portions of $P[x, y]$ hit components of C^D other than C_i^D. In any case, from planarity, $P[x, y]$ must be contained in one region of C_i^D, which it divides into two regions. Let B denote the boundary of that region, and let B_1 and B_2 be the two (closed) parts into which B is divided by $P[x, y]$. Call the region itself R_B. Then let R be the region of C^D (as opposed to just C_i^D) that is bounded by B together possibly with edges from connected components of C^D other than C_i^D, if such are contained in R_B. The region R is divided by $P[x, y]$ into (at least) two subregions R_1 and R_2. The boundary of $R_i, i \in \{1, 2\}$ consists of B_i, $P[x, y]$, and possibly edges from other components. At least one of these two subregions of C^D contains no terminal; without loss of generality we may assume R_1 contains no terminal.

If we add $P[x, y]$ to C^D and remove a subpath of B_1 that contains no vertex of N_i as an interior point, the region R_1 will be merged with a single adjacent region. Because R_1 contains no terminal, the new subset of edges of G^D will still be the dual of a k-terminal cut. We will show that we can always find such a subpath of B_1 having higher weight than $P[x, y]$. This will mean that the replacement yields a k-terminal cut of lesser weight, contradicting the optimality of C. In the three cases below, we use "contains" as a shorthand for "contains as an interior vertex."

Case 1. B_1 contains no vertex of N_i.

This corresponds to Subcase 1.1 of Lemma 2.1 (see Fig. 3a). Because P is a shortest path in G^D, $P[x, y]$ must be the shortest path between x and y in G^D. Thus it is shorter than B_1. Because the latter contains no vertex of N_i, we can replace all of B_1 by $P[x, y]$, thereby obtaining a shorter cut, in contradiction of the assumed optimality of C.

Case 2. B_1 contains vertices of N_i that are in different subtrees of $T^{j-1}[H_i]$.

This corresponds to Subcase 1.2 of Lemma 2.1 (see Fig. 3a). Because B_1 contains vertices from N_i from different subtrees, it has two consecutive such vertices, say a and b. By our ordering of the edges F_h of $T[H_i]$, $w(P[x, y]) \leq w(P) < w(B_1[a, b])$. Thus we can replace $B_1[a, b]$ by $P[x, y]$, obtaining a shorter cut and hence another contradiction.

Case 3. B_1 contains at least one vertex of N_i, and all such vertices are in the same subtree of $T^{j-1}[H_i]$.

This corresponds to Case 2 of Lemma 2.1 (Fig. 3b). Suppose first that none of the vertices from N_i contained in B_1 are in the subtree S of $T_{j-1}[H_i]$ that contains u. Let z be the first vertex of N_i encountered as we traverse B_1 from x to y. Note that z cannot be x by Claim 1. Consider the path from u to z that follows P from u to x and then B_1 from x to z. It connects vertices of N_i in different subtrees, and so $w(P[u, x]) + w(B_1[x, z]) > w(P) > w(P[u, x]) + w(P[x, y])$. Consequently, we must have $w(B_1[x, z]) > w(P[x, y])$, and we can replace $B_1[x, z]$ by $P[x, y]$ for a shorter cut and a contradiction. An analogous argument holds for the case when none of the vertices from N_i contained in B_1 are in the subtree T containing v.

This exhausts the possibilities, so the path corresponding to F_j must be included in C_i^D.

Claim 3. P does not intersect any of the other paths in T_i^D corresponding to edges of $T[H_i]$ except at common endpoints.

Suppose P intersects another path P' of T_i^D at a vertex other than a common endpoint. We shall derive a contradiction, using the fact that Claims 1 and 2 hold for P' as well as for P. Of all the nonendpoint vertices the two paths share, let w be the closest one to u in P. Because all the edges of both P and P' are included in C_i^D by Claim 2, w must have degree three or greater in C_i^D, and hence by definition of *topology*, w must be included in N_i. This, however, contradicts Claim 1, which says that no interior vertices of P are in N_i. Thus Claim 3 holds, and the Lemma is proved. □

Lemma 2.3 treats the connected components C_i^D of C^D in isolation. Let us now look at how they interact. First note that the trees T_i^D are all vertex disjoint, because each is a subgraph of a different connected component of C^D. Thus they constitute a subforest of C^D. Let T^D represent this subforest, the union of all the edges in the T_i^D, and let T be the subset of edges in G whose planar dual is T^D. By Lemma 2.3, the optimal cut C for G will consist of T plus some additional edges, assuming we chose the correct topology for C. To find those additional edges, we delete T from G to obtain a new graph that we shall call $G[0]$. Assuming we have the correct topology, the optimal cut for $G[0]$ will be $C[0] = C - T$.

Let $G[0]^D$ be the embedded dual of $G[0]$ obtained by coalescing all the edges of each T_i^D in G^D to a single point. Note that $G[0]^D$ remains connected and has lost none of the regions from G^D because only trees were coalesced. Assuming that we have the optimal topology, $C[0]^D$ will be a collection of $k-1$ simple cycles in $G[0]^D$, each of which is a biconnected component of $C[0]^D$. That is, a connected component of $C[0]^D$ may consist of more than one cycle, but all such cycles must share a single common vertex. (In particular, the coalesced vertices corresponding to the T_i^D will be such common vertices, although there may be others that were already present in C^D.) See Fig. 6.

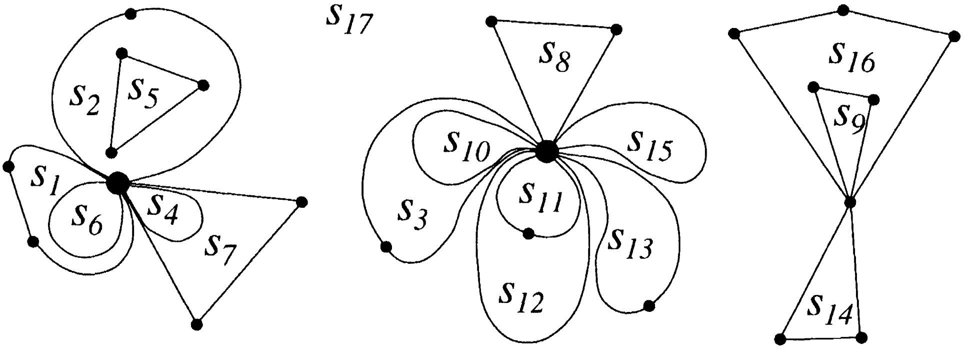


FIG. 6. *The graph $C[0]^D$ obtained by coalescing the spanning trees T_i^D in C^D.*

Note that in the embedding of $G[0]^D$, some of the cycles of $C[0]^D$ will contain others. An innermost cycle will constitute the complete boundary of a region in $C[0]^D$, whereas other regions may have boundaries made up of the edges of several cycles. We shall show that if one treats the cycles in an appropriate "inside out" order, each of these cycles can be viewed as a minimum isolating cut in an appropriately constructed graph. Let us identify each region with the terminal s_i it contains and construct a partial order $\leq$ on the terminals as follows. $s_i \leq s_j$ if and only if the outermost cycle bounding s_i's region is contained in the outermost cycle bounding s_j's region. (By convention, we assume that "the outermost bounding cycle" for the infinite region contains all other cycles, so that the terminal contained in the infinite region is $>$ all other terminals.) Note that by this definition, the outermost bounding cycle for s_i must separate s_i from all terminals $s_j > s_i$.

Let π be an ordering of the terminals that is consistent with this partial order, that is, if $s_i \leq s_j$, then $\pi(i) \leq \pi(j)$. For instance, for the $C[0]^D$ depicted in Fig. 6, such an ordering would be $s_6, s_1, s_4, s_7, s_5, s_2, s_{10}, s_3, s_{11}, s_{12}, s_8, s_{15}, s_{13}, s_9, s_{16}, s_{14}, s_{17}$. We define a sequence of graphs and isolating cuts as follows. For $1 \leq i \leq k-1$, let A_i^D be the set of edges in $G[0]^D$ making up the outer boundary cycle for the region containing terminal $s_{\pi(i)}$. Let A_i be the corresponding set of edges in $G[0]$. Note that $C[0] = \cup_{i=1}^{k-1} A_i$. Now let $G[i]$ be the graph obtained from $G[0]$ by deleting the edges of $\cup_{j=1}^{i} A_j$.

LEMMA 2.4. *For $1 \leq i \leq k-1$, A_i is a minimum isolating cut for terminal $s_{\pi(i)}$ in graph $G[i-1]$.*

Proof. Suppose not, and assume j is the lowest index for which the Lemma is violated. Let B_j be the minimum isolating cut for $s_{\pi(j)}$ in $G[j-1]$. Note that A_j is also an isolating cut for $s_{\pi(j)}$ in $G[j-1]$: by the minimality of j, all terminals $s_{\pi(i)}, i < j$, are already isolated in $G[j-1]$, and by our ordering of the terminals, all terminals of higher index are separated from $s_{\pi(j)}$ by the outermost bounding cycle for $s_{\pi(j)}$, that is, A_j^D. Because the minimum weight isolating cut B_j is unique (by our assumption about edge weights) A_j does not equal it, $w(B_j) < w(A_j)$. But then the set $(C[0] - A_j) \cup B_j$ will have lower weight than $C[0]$. The set will also be a k-terminal cut for $G[0]$. Removing the edges of A_j^D from $C[0]$ will merge $s_{\pi(j)}$'s region with exactly one other, and the terminal for the region must have higher index, by our ordering of the terminals. But B_j separates $s_{\pi(j)}$ from all higher-indexed terminals. Thus $C[0]$ was *not* the optimum cut for $G[0]$, a contradiction. □

Given Lemmas 2.3 and 2.4, the following procedure will handle any given topology $N_1, N_2, \ldots, N_q$ appropriately, either outputting the minimum weight cut with that topology or reporting correctly that the optimal cut cannot have that topology. Assume that as a preprocessing step we have constructed our standard embedding of the dual graph G^D and computed all shortest paths between vertices of G^D.

Procedure check topology $(N_1, N_2, \ldots, N_q)$.

```
1. For $1 \leq i \leq q$, do the following.
   1.1. Construct the auxiliary graph $H_i$, the minimum spanning tree $T[H_i]$, and the
        subtree $T_i^D$.
   1.2. If any two paths in $T_i^D$ corresponding to edges of $T[H_i]$ share a vertex other than
        a common endpoint, reject the topology.
2. If any two subtrees $T_i^D$ share a common vertex, reject the topology.
   Otherwise, let $T^D$ be the union of the edge sets $T_i^D$, let $G[0]$ be the graph obtained
   from $G$ by deleting all edges in the dual set $T$.
3. Let $W^* = \infty$ (our initial estimate of the optimal cut weight).
   For all possible permutations $s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(k)}$ of the terminals, do the following.
   3.1. Set $C = T$.
   3.2. For $1 \leq i \leq k - 1$ do the following:
        3.2.1. Find a minimum isolating cut $A_i$ for $s_{\pi(i)}$ in $G[i-1]$.
        3.2.2. Set $C = C \cup A_i$, and let $G[i]$ be the graph obtained by deleting the edges
               of $A_i$ from $G[i-1]$.
   3.3. If $w(C) < W^*$, set $W^* = w(C)$ and $C^* = C$ (the current best cut with this
        topology).
4. Output $C^*$.
```

It should be clear from the above discussion that this subroutine has the desired properties. Its running time is dominated by that for the minimum isolating cut computations occurring at Step 3.2.1. As discussed, each such computation can be performed using a standard 2-terminal minimum cut algorithm in time $O(n^2 \log n)$, assuming a machine model in which additions, subtractions, and comparisons take constant time. Given our proposed method for imposing the restriction that all sets of edges have unique weights, however, such a model is inappropriate. Even given the standard assumption that the original instance has edge weights that fit into a single computer word, our method for insuring the subset weight uniqueness restriction gives rise to weights whose binary representations involve $\Theta(n)$ bits. With such large numbers, the time bound for the isolating cut computations grows to $O(n^3 \log n)$. We perform $k - 1$ such computations for each of the $k!$ permutations of the terminals, yielding a total of $O(k^k)$ for each topology.

The overall algorithm then consists of performing Procedure Check Topology for each possible topology, of which there are $O((2n)^{2k-4})$ by Lemma 2.2, and outputting the best cut found for any nonrejected topology. Thus we have our claimed result for general fixed k, stated here in slightly more precise form than given in the introduction.

THEOREM 1b. *Given a planar graph G with n vertices and k specified terminals, a minimum k-terminal cut can be constructed in time $O((4k)^k n^{2k-1} \log n)$.*

Note that if we specialize this result to the previously considered case of $k = 3$, the time bound is $O(n^5 \log n)$, substantially larger than the $O(n^3 \log n)$ of Theorem 1a. This is a result of two factors: (1) A factor of n because we can't in general find the needed isolating cuts using planar 2-terminal cut algorithms, as we could when $k = 3$, and so have to use general 2-terminal cut algorithms. (2) A factor of n because we needed to operate with $\Theta(n)$-bit weights to insure that every subset of edges had a unique weight.

It may well be that more efficient algorithms can be derived by careful algorithmic design and analysis, but the bounds we have presented adequately fulfill our goal of showing that the k-terminal cut problem can be solved in polynomial time for fixed k and planar graphs. Moreover, as the NP-completeness result of the next section implies, it is likely that any algorithms for the general problem will have exponential (or at least super-polynomial) running times.

3. Complexity of planar multiterminal cut for arbitrary *k*. We saw in the previous section that the Multiterminal Cut problem is solvable in polynomial time for planar graphs and fixed k. In this and the next section we shall show that the problem becomes NP-hard if we allow either general graphs or general k. In this section we consider the case of planar graphs and general k. As is traditional for complexity results, we concentrate on a decision problem version of our problem. By proving it NP-complete for planar graphs, we imply that the corresponding optimization is NP-hard.

MULTITERMINAL CUT

INSTANCE: Graph $G = (V, E)$, subset $\{s_1, s_2, \ldots, s_k\} \subseteq V$ of terminals, for each edge e a positive integer weight $w(e)$, and a bound B.

QUESTION: Is there a subset $E' \subseteq E$ such that $w(E') \leq B$ and $G' = (V, E - E')$ contains no path linking two distinct terminals?

We shall prove that MULTITERMINAL CUT is NP-complete, even if restricted to bounded-degree planar graphs with all edge weights equal to 1. Note that this is a stronger result than that quoted in the introduction, where for simplicity the question of degree bounds was not raised. We actually prove two separate results. The first allows weights to differ, but restricts vertex degrees to be 3 or less. (Note that the problem becomes trivial if vertex degrees must all be 2 or less.) The second result covers the equal-weight case and follows by a slight modification of the proof of the first.

THEOREM 2a. *MULTITERMINAL CUT is NP-complete for planar graphs, even if edge weights are bounded and the maximum vertex degree is* 3.

Proof. That planar MULTITERMINAL CUT is in NP is immediate. To complete the proof, we need to provide a polynomial transformation to it from some known NP-complete problem. The source problem we choose is PLANAR 3-SATISFIABILITY [9], [19] (PLANAR 3-SAT for short). In the 3-SATISFIABILITY problem we are given a set $X = \{x_1, x_2, \ldots, x_n\}$ of variables and a collection $C = \{c_1, c_2, \ldots, c_m\}$ of 3-element *clauses*, that is, subsets of the set of *literals* for X, where if x_i is a variable, the corresponding literals are x_i and $\overline{x}_i$. The question is whether there exists a satisfying truth assignment for X and C, where a *truth assignment* for X is a subset T of the literals for X that contains precisely one of $x_i, \overline{x}_i$ for each i, $1 \leq i \leq n$, and T satisfies C if for all clauses $c_j \in C$, $c_j \cap T \neq \phi$. (If $x_i \in T$, we say x_i is *true*; if $\overline{x}_i \in T$, we say x_i is *false*.) A natural graph to associate with this problem is the bipartite graph $G_{X,C}$ that has $X \cup C$ as its vertex set and has an edge between the vertices x_i and c_j if c_j contains either of the literals x_i or $\overline{x}_i$. PLANAR 3-SAT is 3-SATISFIABILITY restricted to those instances (X, C) for which $G_{X,C}$ is planar.

For our transformation to planar MULTITERMINAL CUT we shall actually use a restricted variant of PLANAR 3-SAT, one in which we allow clauses of size two as well as three, but require that each variable have degree exactly three in $G_{X,C}$, with one of its literals occurring in two clauses and the other in one. We can prove this variant NP-complete by a simple local-replacement transformation from ordinary PLANAR 3-SAT. Let X, C denote an instance of ordinary PLANAR 3-SAT, as specified above. First note that we may assume without loss of generality that both x_i and $\overline{x}_i$ are contained in $\cup_{j=1}^{m} c_j$ for every variable x_i. If $\overline{x}_i$ does not occur in any clause, we may assume without loss of generality that x_i goes in our truth assignment, and hence all the clauses containing x_i are always satisfied. Thus, we might as well delete those clauses (and the variable x_i) from our instance. Similarly if x_i is not contained in any clause, we might as well delete the variable x_i and all clauses containing $\overline{x}_i$. We may also assume that no clause contains both x_i and $\overline{x}_i$, as such a clause is always satisfied and so could also be deleted. Thus every variable occurs in at least two clauses, and all variable vertices in $G_{X,C}$ have degree at least two.

Our method for converting such an instance X, C to an instance X', C' of our restricted PLANAR 3-SAT variant involves replacing the old variables, adding some new clauses, and modifying the old clauses. First, we replace each variable x_i by a set of $degree(x_i)$ new variables $x_{i,1}, x_{i,2}, \ldots, x_{i,degree(x_i)}$ along with $degree(x_i)$ new clauses. Together, these form a length-$(2degree(x_i))$ cycle in $G_{X',C'}$, as illustrated in Fig. 7 for the case where $degree(x_i) = 5$. The added clauses insure that all the new variables must have the same truth value in any satisfying truth assignment. Note that we make this replacement even for variables with degrees two and three in $G_{X,C}$. These replacements together determine the structure of our new graph $G_{X',C'}$. The construction of X', C' is completed by modifying the old clauses so as to make the new instance consistent with the new graph. For each literal occurrence $x_i(\overline{x}_i)$ in a clause c_j, there must be an edge in $G_{X',C'}$ between c_j and a new variable $x_{i,k}$. We replace the literal $x_i(\overline{x}_i)$ by the new literal $x_{i,k}(\overline{x}_{i,k})$. It is not difficult to verify that the new instance has the desired format and is satisfiable if and only if the original instance was.

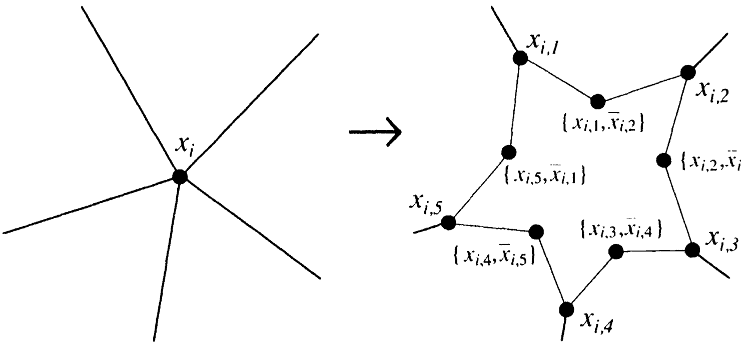

x_i
→
$x_{i,1}$
$x_{i,2}$
$\{x_{i,1},\overline{x}_{i,2}\}$
$\{x_{i,5},\overline{x}_{i,1}\}$
$\{x_{i,2},\overline{x}_{i,3}\}$
$x_{i,5}$
$\{x_{i,3},\overline{x}_{i,4}\}$
$x_{i,3}$
$\{x_{i,4},\overline{x}_{i,5}\}$
$x_{i,4}$

FIG. 7. *Reducing the number of occurrences per variable in PLANAR 3-SAT.*

To complete our proof of Theorem 2a, we now show how to transform an instance X, C of restricted PLANAR 3-SAT into an instance of planar MULTITERMINAL CUT in which all weights are five or less and all vertices have degree three or less. We use the "component design" approach of [9]. Figure 8 shows the components we shall use to represent variables and clauses, with each edge labeled by its weight.

A variable x_i is represented by one of the two structures at the top of the figure, the first if the literal x_i occurs in two clauses, the second if $\overline{x}_i$ occurs in two clauses. (By our definition of restricted PLANAR 3-SAT one of these two possibilities must hold, and the other literal must occur exactly once.) The terminals in these structures are the vertices labeled x_i and $\overline{x}_i$. The *connector triangles* of these structures are the three triangles whose edge weights are 1,1,2 or 1,1,4. The *connector bases* are the weight-2 and weight-4 edges in these triangles, the *connector edges* are the weight-1 edges, and the *connector vertices* are the degree-2 vertices of the triangles. Each connector triangle, base, edge, and vertex will be thought of as representing the literal labeling the terminal nearest to it in the structure. Note that each literal is represented by precisely the same number of connector triangles as it has occurrences in clauses.

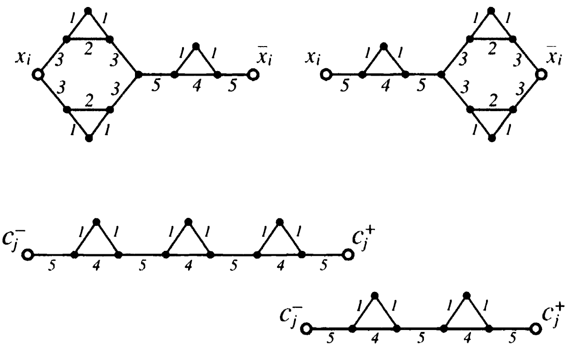


FIG. 8. *Variable and clause components in the proof of Theorem* 2.

A clause c_j is represented by one of the two structures at the bottom of the figure, the first if $|c_j| = 3$, the second if $|c_j| = 2$. Here the terminals are the vertices labeled c_j^- and c_j^+. The *connector triangles* are the triangles whose edge weights are 1,1,4, and the *connector bases, connector edges*, and *connector vertices* are again the weight-4 edges, weight-1 edges, and degree-2 vertices of these triangles. Note that there are precisely as many connector triangles as there are literals in the clause. We shall assume that each connector is assigned to represent a distinct one of those literals.

Note that all the structures of Fig. 8 have maximum degree three. To complete our construction of the MULTITERMINAL CUT instance graph $G[X, C]$, we have only to hook together the clause and variable structures so as to maintain this property and yield a planar graph. This is done by adding *link* edges of weight 2 between variable connector vertices and clause connector vertices as indicated in Fig. 9. (Each clause connector vertex is linked to a variable connector vertex that represents the same literal.) By the definition of restricted PLANAR 3-SAT, there will be precisely the right number of connector vertices for this to be done in a one-one fashion, with all connectors incident on precisely one weight-2 linking edge, implying that the maximum vertex degree in the resulting graph $G[X, C]$ is three. Moreover, it is not difficult to convince oneself that $G[X, C]$ is indeed planar, given that $G_{X,C}$ was.

The only thing remaining to be specified is the upper bound B on the weight of the desired multiterminal cut. This is given by $B = 10n + 4m$, where n and m are the numbers of variables and clauses, respectively, in our restricted PLANAR 3-SAT instance.

The construction just described can clearly be accomplished in polynomial time. To complete the proof that it is a polynomial transformation, we need to show that the following two statements are equivalent.

(1) The PLANAR 3-SAT instance X, C has a satisfying truth assignment.

(2) There is a set E' of edges in $G[X, C]$ whose total weight is B or less and whose deletion would disconnect all $2n + 2m$ terminals from each other.

First suppose that the desired truth assignment T exists. Then the desired multiterminal cut is easy to construct. For each clause c_j pick a literal in $c_j \cap T$ (one must exist because T is a satisfying truth assignment) and delete all three edges of the corresponding connector triangle from the structure representing c_j in $G[X, C]$. For each variable, delete all three edges from each connector triangle representing the literal *not* in T. Finally delete each linking edge that is not adjacent to an already deleted connector triangle. Let E' be the set of deleted edges.

First, we note that E' has the desired cut properties. By our choice of when to delete link edges, we know that at least one endpoint of each remaining link edge must have degree one in the graph after E' is deleted. Thus no path between terminals can pass through a link edge.

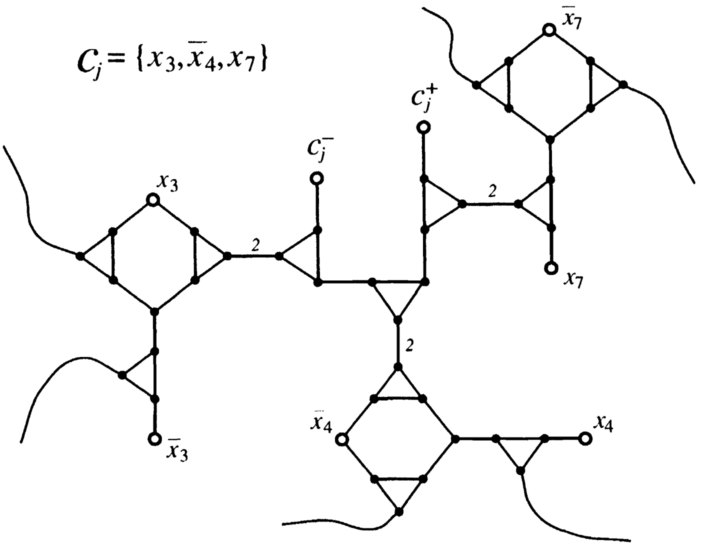


FIG. 9. *Hooking up the components in the proof of Theorem* 2.

Consequently, if there is any path linking terminals, it must either link a pair of terminals x_i, $\bar{x}_i$ in the same variable structure, or a pair c_j^+, c_j^- in the same clause structure. In the former case, all such paths must pass through connector triangles representing both literals, and so must be broken, given that we deleted all connector triangles representing one of the literals. In the latter case, any such path must go through all the connector triangles of the c_j structure, and so must be broken because we deleted one of them.

We now claim that $w(E') = 10n + 4m = B$, and so E' obeys the weight bound and is the desired cut. To see this, let us for accounting purposes divide up the edges of $G[X, C]$ in a slightly different fashion. In particular, let us group each weight-2 link edge with the four weight-1 edges from connector triangles to which it is incident, and call the 5-edge ensemble a *link structure*. See Fig. 10, where edges $\{a, c\}$, $\{b, c\}$, $\{c, d\}$, $\{d, e\}$, and $\{d, f\}$ constitute a link structure joining the structures for clause c_j and variable x_i. (Note that the base edges of the connector triangles, $\{a, b\}$ and $\{e, f\}$, are still viewed as being part of the corresponding clause and variable structures.)

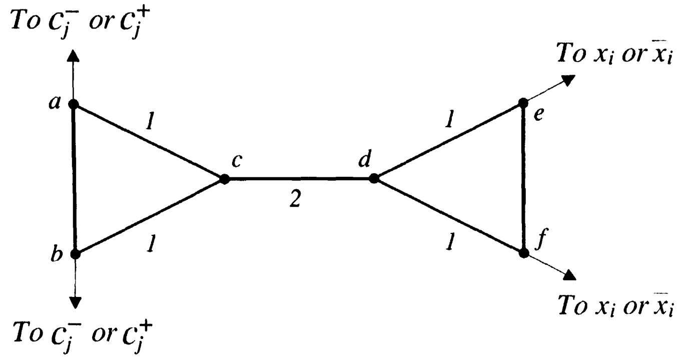


FIG. 10. *A typical connection between a clause and a variable component.*

We claim that for any link structure, the total weight of deleted edges is exactly 2. This is clear if the weight-2 link edge is deleted, because that happens if and only if none of the weight-1 edges were deleted. Suppose one of the pairs of weight-1 edges was deleted. If it is the pair from the clause connector triangle, this means that the corresponding literal must be true. If it is from the variable connector triangle, this means that the corresponding literal must be false. Because both cannot happen simultaneously, only two of the weight-1 edges can be in E'. Thus the total weight of edges deleted from the link structure is two, and the total overall weight of edges deleted from link structures is twice the number of such structures, i.e., $6n$.

Now consider the clause and variable structures. We deleted one edge of weight 4 from each of the former, for a total weight of $4m$. From the latter, we deleted either one edge of weight 4 or two of weight 2, for a total weight of $4n$. This exhausts the possibilities, so we conclude that $w(E') = 6n + 4m + 4n = B$, as desired.

Conversely, suppose a set E' exists with the specified properties. We shall show that X, C has a satisfying truth assignment. We begin with a sequence of "normal form" lemmas. In what follows, we assume E' is a minimum weight set satisfying the specified properties.

LEMMA 3.1. *Suppose e is an edge incident on a degree-3 vertex v. If e has weight greater than or equal to the sum of the weights of the other two edges incident on v, we may assume that e is not in E'.*

Proof. Suppose $e \in E'$, and consider the result of removing e from E' and replacing it by all the other edges incident on v and not already in E'. This will clearly not increase the weight of E'. It also cannot cause any two terminals to become linked by a path in the residual graph. If it did, that path would have to contain e. Because v now has degree one in the residual graph, v would consequently have to be one end of the path, and so would have to be a terminal. This is impossible since, as can be seen from Figs. 7 and 8, no degree-3 vertex in our construction is a terminal. □

As a consequence of Lemma 3.1, we shall assume in what follows that none of the weight-5 edges in any clause structure are in E', and none of the weight-5 and weight-3 edges in any variable structure.

LEMMA 3.2. *If any edge in a cycle of $G[X, C]$ is in E', then at least two are.*

Proof. If e is the only edge from a given cycle in E', then removing e from E' will decrease $w(E')$ without adding any new connections, contradicting our assumption that E' was a minimum weight cut. □

LEMMA 3.3. *At most one connector base (weight-4 edge) in any clause structure is in E'.*

Proof. Suppose the structure for c_j had two weight-4 edges in E'. Let us consider the total clause structure, including all the connector edges, as in Fig. 8. By Lemma 3.2, we may assume that E' contains at least one connector edge adjacent to each connector base it contains, and hence the total weight of edges from the clause structure that are in E' is at least 10. Consider the following modification to E'. Remove one of the two connector bases and add in all the remaining connector edges (there can be at most four). This modification obviously does not increase the weight of E'. Nor can it create any new interterminal paths in the residual graph. Because all the connector edges are now deleted, such a created path could only link c_j^- and c_j^+, but those terminals remain separated since one of the weight-4 edges remains in E'. □

LEMMA 3.4. *For each variable x_i, E' either contains the connector base(s) representing the literal x_i or the base(s) representing the literal $\overline{x}_i$, but not both.*

Proof. Let us consider the total variable structure, including all connector edges, as depicted in Fig. 8. Because no weight-3 and weight-5 edges are in E' by Lemma 3.1, we must either delete the weight-4 base or both weight 2 bases if we are to disconnect terminal x_i from terminal $\overline{x}_i$. By Lemma 3.2, if we delete one of the weight-2 bases, we must delete both. So the only possibility we need to rule out is the situation in which we delete all three base connectors. Should we do this, we will also have to delete at least one connector edge from each connector triangle by Lemma 3.2. Thus the total weight of edges from the structure that would be in E' would be at least 11. Consider the following modification of E': remove the two weight-2 bases and add in all the remaining connector edges (there can be at most three). This modification cannot create any new paths in the residual graph, because x_i remains separated from $\overline{x}_i$ and no path involving a terminal other than these two can traverse the structure. Moreover, it decreases the weight of E' (the total weight of edges from the x_i-structure that are in E' is now 10). Again we have a contradiction of the minimality of E'. □

LEMMA 3.5. *For each link structure, E' contains edges of weight totaling exactly two.*

Proof. We first show that the total must be at least two. This will imply that it is exactly two, because by Lemmas 3.3 and 3.4, E' contains connector bases of total weight at least $4m + 4n$. This leaves at most $6n$ weight for the edges of link structures, and there are $3n$ such structures. So look again at the generic link structure pictured in Fig. 10. By Lemmas 3.1, 3.3, and 3.4, the vertices a and b remain connected to clause terminals in the residual graph, and the vertices e and f remain connected to variable terminals. But this means that there will be a path in the residual graph from a variable terminal to a clause terminal unless we either delete the weight-2 link edge or two weight-1 edges from the same connector triangle. In either case the weight of the two deleted edges is at least two. □

We can now prove our claim that a satisfying truth assignment T must exist. For each variable x_i, put the literal whose connector base is *not* in E' into T. By Lemma 3.4, this is a valid truth assignment. We claim that it satisfies all the clauses. Consider a clause c_j, and let z be the literal corresponding to its broken connector base. By Lemma 3.3, a unique such z exists. Now consider the link structure joining the structure for c_j to the variable structure for z. Note that by our choice of z, the edge (a, b) is in E'. If z were nct in T, then the edge (e, f) would also be in E'. Thus by Lemma 3.2, at least one connector edge from each connector triangle must be in E'. But note that if no other edges from the link structure are in E', there will still be a path from one of a, b to one of e, f within the structure. By Lemmas 3.1, 3.3, and 3.4, this would mean there is a path from a variable terminal to a clause terminal, a contradiction. Thus T is the desired truth assignment, and we have indeed constructed a polynomial transformation, and Theorem 2a is proved. □

THEOREM 2b. MULTITERMINAL CUT *is* NP-*complete for bounded degree planar graphs even if all weights are equal.*

Proof. This is a fairly immediate corollary of the previous proof. Note that our construction would still have worked if we had replaced each edge of weight w by w parallel paths, each consisting of two weight-1 edges. Because no vertex in our construction had incident edges of total weight exceeding 11, this would yield a graph with maximum degree 11, and all edges with weight equal to 1. (The degree bound of 11 is not the best possibility. Using a slight variant on the construction and considerably more complicated arguments, we believe it can be reduced at least to 6.) □

4. Complexity of multiterminal cut for fixed $k \geq 3$ and arbitrary graphs. In this section we prove Theorem 4, which says that MULTITERMINAL CUT is NP-complete for all fixed $k \geq 3$ when arbitrary graphs are allowed. Note that it suffices to prove the problem NP- complete for $k = 3$, because the problem for higher values of k can trivially be derived from that for $k = 3$: simply add $k - 3$ additional terminals with no edges incident on them. Thus we shall concentrate on the $k = 3$ case. In addition to proving this case NP-complete, we shall also present results that may be of assistance in coping with the NP- completeness. We present efficient algorithms that should allow us to reduce significantly the number of vertices and edges in 3-terminal (and multiterminal) cut instances that arise in practice, and we show how to get within a factor of 4/3 of the optimal 3-terminal cut (and $2 - 2/k$ of the optimal k-terminal cut) with a relatively simple heuristic.

First, however, a brief digression into what at first seemed like a promising algorithmic approach to the 3-terminal cut problem, on the basis of results on submodular set functions by Grötschel, Lovász, and Schrijver [14]. The key "gadget" in the NP-completeness proof we shall be presenting also serves as a counterexample to the applicability of this approach. (Indeed, before we discovered the gadget, we already had a proof that either 3-terminal cut was solvable in polynomial time by the submodular set function approach or it was NP-complete by a construction like the one given here.)

4.1. Submodular set functions: algorithms and counterexamples. To understand the results of [14], we first need some definitions. Let U be a finite set. A function f defined on the subsets of U is *submodular* if for any two subsets X and Y of U, $f(X) + f(Y) \geq f(X \cap Y) + f(X \cup Y)$. Grötschel, Lovász, and Schrijver [14] show that if a submodular set function f can be computed in polynomial time, then f can also be *minimized*, that is, a set Y with $f(Y) = \min\{f(X) : X \subseteq U\}$ can be found, in polynomial time. (The algorithm involves an appropriate application of the ellipsoid method.)

A paradigmatic example of a submodular set function involves the usual (2-terminal) minimum cut problem. In this case, U is the set of nonterminal vertices $V - \{s_1, s_2\}$, and $f(X)$ is the total cost of the edges that have precisely one endpoint in the set $X \cup \{s_1\}$. The submodularity of this function is easy to verify, as is the fact that $\min\{f(X) : X \subseteq U\}$ is the weight of a minimum 2- terminal cut. We of course already know how to minimize this function f in polynomial time without resorting to the ellipsoid method, but the formulation is suggestive. Could it be possible that 3-terminal cuts might also be computable as minima of a submodular set function?

It is easy to define a set function for 3-terminal cuts that is analogous to the one above for the 2-terminal case. Let $U = V - \{s_1, s_2, s_3\}$ be the set of nonterminal vertices, and for any subset X of U, let $f(X)$ be the minimum cost of a 3-terminal cut that leaves no vertex in X connected to s_2 or s_3 (and no vertex in $U - X$ is connected to s_1 or a vertex in X). It is easy to see that the minimum value for this function equals the minimum weight for *any* 3-terminal cut. Moreover, we can use a polynomial-time algorithm for the 2-terminal cut problem to evaluate f in polynomial time. Given a subset X of U, find a minimum weight cut separating s_2 from s_3 in the graph obtained by deleting s_1 and all the vertices in X from G. Add to the weight of this cut the weight of all edges with precisely one endpoint in $X \cup \{s_1\}$. Therefore, if f were submodular (for all graphs), we could solve the 3-terminal cut problem in polynomial time.

Unfortunately, this is not the case. Consider the 9-vertex graph C depicted in Fig. 11. Note that in addition to the three terminals s_1, s_2, s_3, the graph contains two specified vertices x and y. The 12 edges incident on the terminals have weight 4, as indicated in the figure. The other 6 edges, unlabeled in the figure, have weight 1. Let c^* be the cost of an optimal 3-terminal cut for C. For each i, j, $1 \leq i, j \leq 3$, let an *i, j-cut* be a 3-terminal cut that leaves vertex x connected to s_i and vertex y connected to s_j, and let $c(i, j)$ be the cost of a minimum *i, j-cut*. The sets X and Y that cause f to violate submodularity are defined as follows:

Let X be the set of vertices connected to s_1 in an optimal 1,2 cut. (Note that by definition of *i, j-cut*, x is in X and y is not.) Let Y be the set of vertices connected to s_1 in an optimal 2, 1 cut. (Note that y is in Y and x is not.) By definition, we have $f(X) = c(1, 2)$ and $f(Y) = c(2, 1)$. We also must have $f(X \cup Y) \geq c(1, 1)$ and $f(X \cap Y) \geq \min\{c(2, 3), c(3, 2), c(2, 2), c(3, 3)\}$.

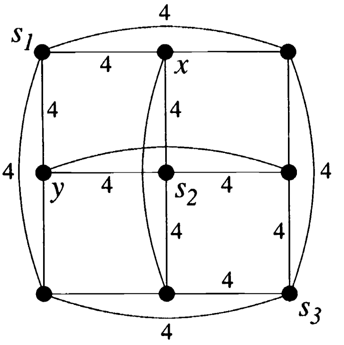


FIG. 11. *Graph C: submodularity counterexample and* NP- *completeness gadget.*

Thus if f were to be submodular, we would need to have $c(1, 2) + c(2, 1) \geq c(1, 1) + \min\{c(2, 3), c(3, 2), c(2, 2), c(3, 3)\}$. In light of the following lemma, however, this claim is false.

LEMMA 4.1. *For the graph C of Fig.* 11, *the following properties hold:*

(a) $c(1, 2) = c(2, 1) = c^*$,

(b) $c(i, j) \geq c^* + 1$ *for all other pairs* i, j, *and*

(c) $c(1, 1) = c(2, 2) = c^* + 1$.

Proof. As depicted in Fig. 11, graph C has its vertices located at the nodes of a 3 × 3 grid. As an alternative naming convention for the vertices, let v_{ij} denote the vertex in the row i, column j, $1 \leq i, j \leq 3$. Thus the terminals are $s_1 = v_{11}, s_2 = v_{22}$, and $s_3 = v_{33}$, and the distinguished vertices x and y are v_{12} and v_{21}, respectively.

We first claim that $c(1, 2) = c(2, 1) = c^* = 27$. Note that the set consisting of the nine vertical edges in C is a weight-27 1,2 cut in which every vertex v_{ij} is left connected with the terminal s_i in its row, $1 \leq i \leq 3$. Similarly, the set consisting of the nine horizontal edges is a weight-27 2,1-cut in which every vertex v_{ij} is connected to the terminal s_j in its column. Could there be a 3-terminal cut of less weight? Note that each of the six nonterminal vertices v_{ij} is connected by weight-4 edges to both s_i and s_j. Thus in any 3-terminal cut at least six weight-4 edges must be deleted, one for each nonterminal v_{ij}. If the weight of the cut is to be 27 or less, no further weight-4 edges can be deleted, and so each nonterminal must remain connected to one terminal by a weight-4 edge. In particular, v_{ij} must remain connected either to s_i or s_j. Let us say that v_{ij} *belongs* to the terminal to which it remains connected.

Now consider the weight-1 edges. These all join nonterminal vertices, and together they form a cycle of length six with alternating vertical and horizontal edges: $v_{21}(y)$-v_{31}-v_{32}-$v_{12}(x)$-v_{13}-v_{23}-v_{21}. Suppose two consecutive edges of this cycle remain undeleted, say without loss of generality $\{v_{ij}, v_{ik}\}$ and $\{v_{ik}, v_{lk}\}$. Note that we must have $i \neq j, i \neq k, j \neq k, i \neq l$, and $l \neq k$. In order to rule out any interterminal paths, v_{ij}, v_{ik}, and v_{lk} must then all belong to the same terminal. That terminal must be either s_i or s_j, because those are the only two to which v_{ij} can belong. Similarly, it must be either s_i or s_k, because those are the only two to which v_{ik} can belong. Thus, because $j \neq k$, it must be i. But this is impossible, because v_{lk} can only belong to s_l or s_k, and neither of these can equal i, as already observed. Thus the cut must contain one of every two consecutive edges in the cycle of weight-1 edges, or at least three such edges, for a total weight of at least 27. The only way that precisely three can be chosen is if we take every other edge in the cycle, that is, either the three vertical edges or the three horizontal ones, as was done in the horizontal and vertical cuts mentioned in the previous paragraph. We now can conclude that those cuts were optimal, and hence $c^* = c(1, 2) = c(2, 1) = 27$. Thus part (a) of the lemma holds.

For part (b), we shall show that any cut other than the horizontal and vertical cuts mentioned above must have weight 28 or more. Assume there were such a cut of weight 27. As argued above, its intersection with the cycle of weight-1 edges is either the set of three vertical edges of the cycle or the set of three horizontal ones. By the symmetry of C, we may assume that it contains the three vertical edges. Because it is not the vertical cut, it must thus fail to contain at least one vertical weight-4 edge. For specificity (and again without loss of generality because of the symmetry of C), we may assume that the missing vertical weight-4 edge is $\{s_1, v_{21}\}$. But then, since the horizontal weight-1 edge $\{v_{21}, v_{23}\}$ is also not in the cut, the vertex v_{23} is connected to the terminal s_1 in the residual graph. This means that the cut must contain both the edge joining v_{23} to s_2 and the edge joining it to s_3. Because, as remarked earlier, the cut must contain at least one weight-4 edge incident on each of the six nonterminals, this yields a total of at least seven weight-4 edges, and so the cut will have weight at least 28, a contradiction. Thus part (b) is proven.

To prove part (c), we need to show that there exist 1,1- and 2,2-cuts of weight 28. It suffices (again by symmetry) to consider the 1,1 case. A weight-28 1,1-cut is obtained by modifying the weight-27 cut consisting of all vertical edges. Instead of deleting the vertical weight-4 edge $\{y, s_1\}$, we delete the two horizontal edges incident on y: the weight-4 edge $\{y, s_2\}$ and the weight-1 edge $\{y, v_{23}\}$. It is easy to see that the result continues to be a 3-terminal cut, although it now has weight 28, and y as well as x is connected to s_1. Thus the cut is a 1,1-cut of the desired weight, and part (c) holds. □

4.2. The NP-completeness of 3-terminal cut. We are now ready to prove Theorem 3, which we restate in terms of the $k = 3$ special case to which it reduces.

THEOREM 3. *If arbitrary graphs are allowed, MULTITERMINAL CUT for $k = 3$ (i.e., 3-TERMINAL CUT) is NP-complete even if all weights are equal to* 1.

Proof. It is immediate that 3-TERMINAL CUT is in NP. We shall show that 3-TERMINAL CUT is NP-complete if weights 1 and 4 are allowed. This will suffice to prove the theorem, because a weight-4 edge could always be replaced by four parallel length-2 paths of weight-1 edges.

Our proof is by a polynomial transformation from the SIMPLE MAX CUT problem [9], [10]. In SIMPLE MAX CUT, we are given a graph $G = (V, E)$ (without weights) and a number K, and are asked whether there is a partition of the vertices of G into two sets V_1 and V_2 such that there are at least K edges between V_1 and V_2. Given (G, K), we construct a corresponding instance (F, B) of 3-TERMINAL CUT as follows.

The graph F has three terminals s_1, s_2, s_3 and contains the vertices of G, but not the edges. For each edge $\{u, v\}$ of G, the graph F instead contains a copy of our "gadget graph" C, with the vertices s_1, s_2, s_3 of C identified with their named counterparts in F, and with x and y identified with u and v. The other vertices of different copies of C (i.e., the four nonterminals other than x and y) are distinct. Thus the total number of vertices in F is $3 + |V| + 4|E|$. We claim that G has a cut of size K or greater if and only if F has a 3-terminal cut of weight $B = 28|E| - K$ or less. Given that F and B can clearly be constructed in polynomial time, Theorem 3 will follow from this claim.

So suppose there is a cut V_1, V_2 for G of size $K' \geq K$. Consider the 3-terminal cut induced by the following assignment of the vertices of F to the terminals: First, vertices in V_1 are assigned to s_1, and vertices in V_2 are assigned to s_2. At this point, each copy of C has its x and y vertices assigned to one of s_1 or s_2, say s_i for x and s_j for y. Assign the remaining nonterminals of this copy of C to terminals according to a minimum weight i, j-cut for C. Note that the contribution to our 3-terminal cut from edges in this copy of C will thus be $c(i, j)$. In particular, if x and y were in the same set the contribution will by Lemma 4.1 be 28; if they were in different sets it will be 27. The overall weight thus becomes $28|E| - K' \leq 28|E| - K$, as desired.

Conversely, suppose a 3-terminal cut of size $B' \leq 28|E| - K$ exists. Let V_i be the set of vertices in V that are left connected to s_i, $i = 1, 2, 3$. For each edge $\{u, v\}$ of G with $u \in V_i$, $v \in V_j$, the cut removes edges of total weight at least $c(i, j)$ from the corresponding copy of C. By Lemma 4.1, $c(i, j) \geq 28$ unless $\{i, j\} = \{1, 2\}$, and so there must be at least K edges between V_1 and V_2. Thus we can assign the vertices of V_3 to the two sets V_1 and V_2 arbitrarily and still obtain the desired cut for G. □

Note that the graph constructed in our proof of Theorem 3 does not have bounded vertex degrees. This is unavoidable, so long as we assume all edge weights are equal. If k is fixed, all edge weights are equal, and there is a bound d on vertex degree, then multiterminal cut can be solved in polynomial time! Observe that in this case the weight of a cut is simply the number of edges it contains, and an optimal cut can contain no more than kd edges (because the cut that simply breaks all the edges incident on each terminal is no bigger than this). But because k and d are fixed, kd is a constant independent of n. Consequently, we can use exhaustive search and still take time that is polynomially bounded in n.

If we remove the restriction to equal-weight edges, however, the fixed-k problem becomes NP-complete even if all vertex degrees are three or less. Simply replace each vertex v in our construction that has $degree(v) > 3$ by a cycle of $degree(v)$ vertices, each of which is adjacent to one of the former neighbors of v, and let the weights of the cycle edges be sufficiently high that they can't be chosen for an optimal cut.

4.3. Reducing the instance size. The results of §§4.1 and 4.2 effectively dash any hope of finding optimal k-terminal cuts, $k \geq 3$, efficiently by means of 2-terminal cut (i.e., max flow) algorithms. Such algorithms may still be useful, however, as we shall see in this and the next section. Recall that an *isolating cut* for a terminal s_i is a set of edges that separates s_i from the other terminals, and that minimum weight isolating cuts can be found in $O(nm \log(n^2/m))$ time by performing max flow computations in a modified graph. The following lemma implies that the computation of minimum weight isolating cuts can be used to reduce the number of vertices in an instance. Suppose $G = (V, E)$ is a connected graph with specified terminals $s_1, s_2, \ldots, s_k$.

LEMMA 4.2. *Suppose $i \in \{1, 2, \ldots, k\}$, $k \geq 3$. Let E_i be a minimum weight isolating cut for terminal s_i, and let V_i be the set of vertices that remain connected to s_i when the edges of E_i are removed from G. Then there exists an optimal k-terminal cut for G that leaves all the vertices of V_i connected to s_i.*

Proof. Without loss of generality, we may assume that $i = 1$. Suppose the lemma were false for some $k \geq 3$, and let G be a minimal counterexample, that is, a counterexample having the fewest vertices, and, among those counterexamples with that number of vertices, the fewest edges. Let E^* be a minimum weight k-terminal cut for G, and let c_1 and c^* be the weights of E_1 and E^*, respectively. Label every vertex u of G with a pair (j, l) where $j = 1$ or 2 depending on whether u is in V_1 or not, and l is the index of the terminal to which u is left connected under E^*. (This index is defined, because u must be left connected to some terminal if E^* is to have minimum weight.)

The above labelling partitions the vertices into $2k$ groups. We first claim that, by the minimality of G, each group can consist of at most one vertex. Suppose u and v have the same label, and consider the graph G' in which we merge u and v. The cuts induced on G' by E_1 and E^* will have the same weights as did the original cuts, and clearly no better cuts of the corresponding types can exist. Thus G' would be a counterexample with fewer vertices, a contradiction of our minimality assumption.

Next we claim that if $\{u, v\}$ is an edge of G and the labels of u and v are (j_u, l_u) and (j_v, l_v) respectively, then it cannot be the case that both $j_u \neq j_v$ and $l_u \neq l_v$. Suppose such an edge e were in G, and note that e must be in both E_1 and E^*. Thus, consider the graph G'

obtained by removing e from G. The sets $E_1 - \{e\}$ and $E^* - \{e\}$ will continue to be isolating and k-terminal cuts respectively for G' and the weight of each will drop by $w(e)$. Furthermore, there cannot be a k-terminal cut E' of weight less than $c^* - w(e)$ (isolating cut E_1' of weight less than $c_1 - w(e)$) in G', as otherwise we could get a k-terminal cut in G of weight less than c^* (isolating cut of weight less than c_1) by simply adding e to E' (to E_1'). Thus $E_1 - \{e\}$ and $E^* - \{e\}$ continue to be optimal in G', so that G' would be a counterexample with fewer edges and the same number of vertices, again a contradiction of our minimality assumption.

Thus for $k = 3$ our minimal counterexample is a subgraph of the 6-vertex graph depicted in Fig. 12, where the vertices of G are placed on a 2×3 grid, with the vertex in row i and column $j(v_{ij})$ being the one with label (i, j). For $k > 3$, we would have a $2 \times k$ grid, with all k vertices in the top row connected in a clique, as are all k vertices in the bottom row. Note that for G to be a counterexample, at least one of the nonterminal vertices in the graph must be present in G. The set V_1 of vertices connected to s_1 under E_1 is simply the set of all vertices in the top row that are present in G.

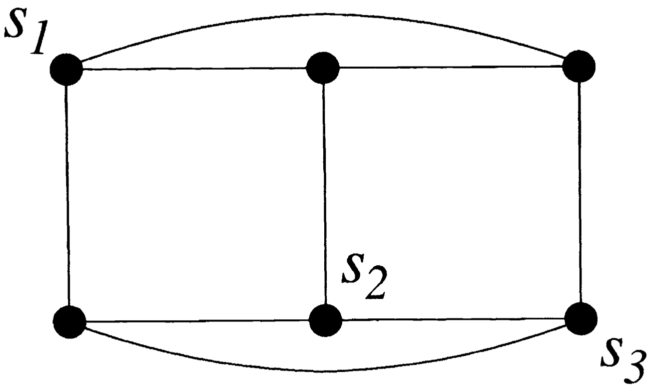


FIG. 12. *Schematic of the assumed minimal counterexample in Lemma* 4.2 (*if* $k = 3$).

Now observe that because E_1 consists of the vertical edges in this figure, these edges have total weight c_1. Similarly, E^* consists of the horizontal edges, and these edges consequently have total weight C^*. Thus the total weight of all the edges in G is $c_1 + c^*$. Consider the set $\hat{E}_1$ of edges incident on s_1 in G. This is clearly an isolating cut, and so $w(\hat{E}_1) \geq c_1$. But then consider the k-terminal cut $\hat{E}^*$ consisting of all edges incident on s_2 through s_k. This is disjoint from $\hat{E}_1$, and so can have weight at most $c^* + c_1 - w(\hat{E}_1) \leq c^*$. Hence it has weight precisely c^* and is itself an optimal k-terminal cut. But note that it leaves s_1 connected to all the vertices in V_1. Thus G actually satisfies the lemma statement, and there can be no counterexample. □

Using Lemma 4.2, we can reduce the number of vertices in our instance by $|V_i - 1|$. Simply construct a new graph in which all the vertices of V_i are merged into the terminal s_i. An optimal k-terminal cut for this shrunken graph will induce an optimal cut for the original one.

To get the maximum effect from applying Lemma 4.2 in this way, we should start with the minimum isolating cut for s_i such that V_i is as big as possible. Let us say that a set V_i is an *isolation set* for a terminal s_i if it is the set of vertices left connected to s_i by some minimum weight isolating cut. Let us call it an *optimum isolation set* for s_i if it has maximum cardinality over all isolation sets for s_i. From observations made in [7, pp. 10–13], it can be seen that the optimum isolation set for a given terminal is unique and contains all other isolation sets for that terminal. It can be found by performing one maximum-flow computation followed by some linear-time post-processing. A corollary of the following Lemma is that k optimum isolating set computations suffice to shrink G as far as it can go.

LEMMA 4.3. *Let V_i be the optimum isolation set for s_i, $1 \leq i \leq k$, and let $\overline{G}$ be the graph obtained from G by merging all the vertices of V_1 into s_1. Then in $\overline{G}$ the optimum isolation set for s_1 is $\{s_1\}$, and the optimum isolation set for s_i is $V_i - V_1$, $2 \leq i \leq k$.*

Proof. For $i = 1, 2, \ldots, k$, let w_i and $\overline{w}_i$ denote the weights of a minimum weight isolating cut for s_i in G and $\overline{G}$, respectively. Let $\overline{V}_i$ be the optimum isolation set for s_i in $\overline{G}$.

We begin with $\overline{V}_1$. Note that the isolating cut for s_1 that separates s_1 from all other vertices in $\overline{G}$ has weight w_1 by hypothesis, and we must have $\overline{w}_1 \leq w_1$. On the other hand, $\overline{V}_1 \cup V_1$ induces an isolating cut for s_1 in G that has weight $\overline{w}_1$, so we must have $\overline{w}_1 \geq w_1$. Thus equality holds and $\overline{V}_1 \cup V_1$ is an isolation set for s_1 in G. But this means that $\overline{V}_1 \cup V_1$ is contained in V_1, by the properties of maximum isolation sets mentioned above, and hence $\overline{V}_1 \subseteq V_1$. Because the only member of V_1 that exists in $\overline{G}$ is s_1, this implies that $\overline{V}_1 = \{s_1\}$, as claimed.

Now consider $\overline{V}_2$. (The arguments for $\overline{V}_i, k > 2$ are analogous and hence omitted.) Let $\hat{V}_2 = V_2 - V_1$. We first show that $\hat{V}_2$ induces a minimum weight isolating cut for s_2 in G (although $\hat{V}_2$ will not be an optimum isolation set for s_2 in G unless $V_1 \cap V_2 = \phi$). Partition the vertices of V into the four sets $A = V_1 - V_2$, $B = V_2 - V_1$, $C = V_1 \cap V_2$, and $D = V - (V_1 \cup V_2)$. If X and Y are two disjoint subsets of the vertices of G, let $E(X; Y)$ denote the set of edges that link vertices in X with vertices in Y, and let $w(X; Y)$ denote their weight.

By hypothesis $w_1 = w(A; B) + w(A; D) + w(B; C) + w(C; D)$ and $w_2 = w(A; B) + w(B; D) + w(A; C) + w(C; D)$. Thus $w_1 + w_2 = W + w(A; B) + w(C; D)$, where W is the total weight of the edges that link vertices in different sets of our partition. Now consider the isolating cuts for s_1 and s_2 induced by $V_1 - V_2$ and $V_2 - V_1$, respectively, and let $\hat{w}_1$ and $\hat{w}_2$ be their weights. Then $\hat{w}_1 = w(A; B) + w(A; C) + w(A; D)$ and $\hat{w}_2 = w(A; B) + w(B; C) + w(B; D)$, so that $\hat{w}_1 + \hat{w}_2 = W + w(A; B) - w(C; D) \leq w_1 + w_2$. Because by hypothesis we must have $w_i \leq \hat{w}_i$, $i \in \{1, 2\}$, this implies that equality must hold in both cases, and so $\hat{V}_2 = V_2 - V_1$ must induce a minimum weight isolating cut for s_2 in G. Because none of the vertices of $\hat{V}_2$ were merged with s_1 in the construction of $\overline{G}$, $\hat{V}_2$ must consequently also induce a minimum weight isolating cut for s_2 in $\overline{G}$, and so $\hat{w}_2 = \overline{w}_2 = w_2$. Thus $\hat{V}_2$ is an isolation set for s_2 in $\overline{G}$, and by the properties of optimal isolation sets, we must have $\hat{V}_2 = V_2 - V_1 \subseteq \overline{V}_2$.

On the other hand, because $\overline{V}_2$ induces an isolating cut of weight $\overline{w}_2 = w_2$ for s_2 in $\overline{G}$, it induces an isolating cut of the same weight in G, and so $\overline{V}_2$ is an isolation set for s_2 in G. Thus we must have $\overline{V}_2 \subseteq V_2$. Because the only vertices of V_2 that remain in $\overline{G}$ are those in $V_2 - V_1$, this implies that $\overline{V}_2 = V_2 - V_1$, as claimed. This completes the proof of the lemma. □

Lemma 4.3 indicates both the efficiency with which we can apply Lemma 4.2 to reduce the instance size, and the bounds on how much shrinkage can be obtained. In particular, k optimum separation set computations suffice to yield all the shrinkage one can expect. Note that the proof of Lemma 4.3 would apply just as well if we renamed the terminals in any order. So let $G_0 = G$, and inductively obtain G_i from G_{i-1} by performing an optimum isolation set computation for s_i and merging all the vertices in the set obtained into the terminal s_i. Lemma 4.2 says that an optimal k-terminal cut in G_i induces one in G_{i-1} (and, by induction, in G), and Lemma 4.3 says that the optimum isolation set for s_i in G_i is $\{s_i\}$ (and by induction, the optimum isolation set for s_h, $1 \leq h \leq i$, is $\{s_h\}$). Thus in G_k, the optimum isolation set for each terminal consists of the terminal itself, but a minimum weight k-terminal cut still induces one in the original graph G. Thus G_k is a maximally shrunken graph that can still induce an optimal k-terminal cut.

4.4. Near-optimal multiterminal cuts. If one is willing to settle for cuts that are only *near*-optimal, one can exploit a bit further our ability to construct optimum isolating cuts. Consider the following straightforward heuristic.

Isolation heuristic.

1. For $1 \leq i \leq k$ construct a minimum weight isolating cut $\hat{E}_i$ for terminal s_i.
2. Determine h such that $\hat{E}_h$ has maximum weight among all the $\hat{E}_i$'s.
3. Let $\hat{E}$ be the union of all cuts $\hat{E}_i$ except $\hat{E}_h$.
4. Return $\hat{E}$.

Note that the Isolation Heuristic clearly outputs a k-terminal cut. Moreover, it can be implemented to run in $O(knm \log(n^2/m))$ time by using the max flow algorithm of [12] to compute each of the k required isolating cuts. This is the heuristic to which we referred in Theorem 4 of the Introduction. A more precise statement of that theorem can now be given.

THEOREM 4. *The Isolation Heuristic constructs a k-terminal cut whose weight is guaranteed to be no more than* $2(k-1)/k$ *times the optimal weight.*

Proof. Let $\overline{E}$ be an optimal k-terminal cut, and let $\overline{w} = w(\overline{E})$. For $1 \leq i \leq k$, let $\overline{V}_i$ be the set of vertices left connected to s_i by $\overline{E}$, and let $\overline{E}_i$ be the set of edges in $\overline{E}$ with one endpoint in $\overline{V}_i$. Note first that for each i, the set $\overline{E}_i$ is an isolating cut for s_i. Hence $w(\overline{E}_i) \geq w(\hat{E}_i)$. On the other hand, each edge in $\overline{E}$ is in exactly two different sets $\overline{E}_i$, and so $\Sigma_{i=1}^{k} w(\overline{E}_i) = 2\overline{w}$. Thus

$$w(\hat{E}) \leq \frac{k-1}{k}\sum_{i=1}^{k} w(\hat{E}_i) \leq \frac{k-1}{k}\sum_{i=1}^{k} w(\overline{E}_i) = 2\frac{k-1}{k}\overline{w}$$

as claimed. □

The bound given by Theorem 4 is tight. More precisely, for each $k \geq 3$ and any $\varepsilon > 0$, there exist instances for which the cut constructed by the Isolation Heuristic has weight $(2-\varepsilon)(k-1)/k$ times the optimal. The generic construction contains $2k$ edges and vertices, with the nonterminal vertices $v_1, v_2, \ldots, v_k$ linked in a simple k-cycle of weight-1 edges, and terminal s_i linked by a weight-$(2-\varepsilon)$ edge to vertex v_i, $1 \leq i \leq k$. See Fig. 13 for an illustration of the case for $k = 4$.

Note that for each terminal s_i, the minimum weight isolating cut is unique and consists of the edge of weight $2-\varepsilon$ connecting it to v_i. Thus the weight of the cut constructed by the heuristic is $(k-1)(2-\varepsilon)$. An optimal cut, on the other hand, would consist simply of all the k weight-1 edges in the cycle linking the nonterminals, for a total weight of k. The ratio thus has the claimed value. (Note that if we are willing to assume that our heuristic always breaks ties in the worst possible way, we can take $\varepsilon = 0$ in the above example and obtain a ratio that precisely matches the upper bound of Theorem 4.)

For $k = 3$, the ratio guaranteed by Theorem 4 is 4/3 and fairly close to 1. An interesting question is whether there are any polynomial time heuristics that provide better guarantees. In particular, is it conceivable that we could guarantee ratios arbitrarily close to 1? Formally, is there a *polynomial time approximation scheme* for 3-terminal cut i.e., a sequence of polynomial time algorithms A_t, where A_t is guaranteed to find a 3-terminal cut of weight at most $1 + 1/t$ times the optimal weight? (The algorithms need not be polynomial in t, only in the instance size.)

There is significant evidence that the answer to this question is no. In particular, our proof of Theorem 3 implies that the 3-terminal cut problem is what is known as *MAX SNP-hard* [21], and this implies that 3-terminal cut cannot have a polynomial time approximation scheme unless P = NP. Let us explain.

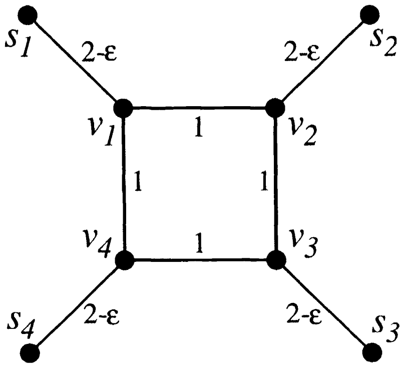


FIG. 13. *Worst-case graph for the Isolation Heuristic when* $k = 4$.

An optimization problem is in MAX SNP if the optimal value sought can be described as $\max_S |\{\bar{x} : \phi(\bar{x}, I, S)\}|$, where I is an instance, S is a relational structure such as a truth assignment in the case of 3-SAT, x is the object being counted, such as satisfied clauses, and ϕ is a first order logical expression. A second relevant example of a problem in the class MAX SNP is *Max Cut*, the optimization version of the problem SIMPLE MAX CUT that we used to prove that 3-TERMINAL CUT is NP-complete in §4.2. In the Max Cut problem, the instance I is a graph, the structure S is a partition of the vertices into two sets, and the objects $\bar{x}$ are edges, where $\phi(\bar{x}, I, S)$ is true if and only if $\bar{x}$ does not have both its endpoints in the same set of the partition S.

MAX SNP was introduced in [21], where it was observed that although each problem in MAX SNP could be approximated to within *some* constant ratio in polynomial time, no problem in the class was known to have a polynomial time approximation scheme. The concept of MAX SNP-hardness was also introduced in [21] and defined in such a way that it could be applied both to problems, such as MAX 3-SAT and Max Cut, that are in MAX SNP and to problems, such as 3-Terminal Cut, that are minimization problems and hence by definition not in MAX SNP. The key observation about MAX SNP-hardness in [21] was that no MAX SNP-hard problem could have a polynomial time approximation scheme unless all problems in MAX SNP had them. Arora et al. [1] have now shown that the latter event cannot occur unless $P = NP$. Consequently no MAX-SNP-hard problem can have a polynomial time approximation scheme unless $P = NP$.

Proofs of MAX SNP-hardness involve *linear reductions*, a generalization of the familiar polynomial transformations used in NP-hardness proofs. Let A and B be two optimization problems (either maximization or minimization). We say that A linearly reduces to B if there are two polynomial time algorithms f and g and constants $\alpha, \beta > 0$ such that

1. Given an instance a of A, algorithm f produces an instance $b = f(a)$ of B such that the cost of an optimal solution for b, $opt(b)$, is at most $\alpha \cdot opt(a)$, and

2. Given a, $b = f(a)$, and any solution y of b, algorithm g produces a solution x of a such that $|cost(x) - opt(a)| \le \beta |cost(y) - opt(b)|$.

It can be shown [21] that linear reductions are transitive, that is, are closed under composition, and that if A linearly reduces to B and B has a polynomial-time approximation algorithm guaranteed to produce solutions y with $|cost(y) - opt(b)|/opt(b) \le \varepsilon$, then A has a polynomial-time approximation algorithm that guarantees to keep the analogous ratio bounded by $\alpha\beta\varepsilon$. Thus if B had a polynomial time approximation scheme, so would A. A problem is MAX SNP-hard if every problem in MAX SNP linearly reduces it to, or equivalently, given the transitivity of linear transformations, if some single previously identified MAX SNP-hard problem linearly reduces to it.

THEOREM 5. *For any fixed $k \geq 3$, k-terminal cut is MAX SNP-hard.*

Proof. We prove the result for $k = 3$. The extension to $k > 3$ follows immediately. Our proof is by a linear reduction from the Max Cut problem described above, previously proved MAX SNP-hard in [21]. For the reduction, we need only reinterpret the transformation from SIMPLE MAX CUT to 3-TERMINAL CUT used in the proof of Theorem 3. Note that for that construction we showed that if k is the size of the maximum cut in the original instance, then the size of the optimal 3-terminal cut is $28|E| - K$. Now note that the maximum cut must have size at least $|E|/2$, because a simple greedy heuristic will construct a cut that large. (Start with two adjacent vertices as the nuclei of V_1 and V_2, and then assign the rest of the vertices one at a time, choosing for each the set that maximizes the number of edges added to the cut.) Thus if we let $f(G)$ denote the instance of 3-Terminal Cut derived from an instance G of Max Cut, we have $OPT_{\text{3-TerminalCut}}(f(G)) \leq 56\, OPT_{\text{MAXCUT}}(G)$, and our transformation satisfies Property (a) of the definition of linear reduction with $\alpha = 56$.

For property (b), note that our proof of Theorem 3 implies that any solution y for $f(G)$ of weight $28|E| - K$ can be easily converted to a solution $x = g(y)$ of size K for G. Thus for any solution y of $f(G)$ we have $|cost(x) - opt(a)| \leq |cost(y) - opt(b)|$, and so property (b) holds with $\beta = 1$.

We conclude that the transformation used in the proof of Theorem 3 was a linear reduction, and so 3-terminal cut is MAX SNP-hard. □

5. Related results and open problems. Our NP-completeness results in §3 and §4 can be adapted to several related problems of interest. In 1969, Hu [17] raised the question of the complexity of the following problem that we might call the *multipair cut* problem. Suppose we are given a list of vertex pairs (u_i, v_i), $1 \leq i \leq k$, and are asked to find a minimum $C(u_1, u_2, \ldots, u_k, v_1, v_2, \ldots, v_k)$ cut, that is, a minimum weight set of edges separating each pair of vertices u_i, v_i, $1 \leq i \leq k$. This is just 2-terminal cut when $k = 1$. The problem is also polynomial time solvable when $k = 2$, by using two applications of a 2-terminal cut algorithm [24]. Our result for 3-terminal cut implies that it is NP-hard for arbitrary graphs when $k = 3$, even if all edge weights are equal: merely let the three pairs be (s_1, s_2), (s_2, s_3), and (s_3, s_1). (If one wants all the u_i and v_i to be distinct, the problem remains NP-hard, as can be proved via a simple modification to the input graph.)

Note that for any fixed k the Isolation Heuristic of §4.4 can be used in the design of a polynomial-time approximation algorithm for multipair cut. Consider partitions P of the vertices $u_1, u_2, \ldots, u_k, v_1, v_2, \ldots, v_k$ into sets $S_1, \ldots, S_{|P|}$ such that no pair (u_i, v_i) is in the same set, and such that for every pair of sets S_h and S_j there is some i such that u_i is in one set and v_i is in the other. Note that the latter constraint implies that $|P| \leq \sqrt{2k}+1$, and so there are at most $(\sqrt{2k}+1)^{2k}/(\sqrt{2k}+1)!$ such partitions. For any such partition P, let G_P be the graph obtained by merging all the vertices in the set S_j into a single terminal vertex s_j, $1 \leq j \leq |P|$. Run the Isolation Heuristic on each such graph G_P, at a cost of $O((\sqrt{2k}+1)nm \log(n^2/m))$ per graph, and output the best cut found. Because the optimal $C(u_1, u_2, \ldots, u_k, v_1, v_2, \ldots, v_k)$ cut must induce one of the partitions P, and because no partition contains more than $\sqrt{2k} + 1$ sets, the weight of the cut we output is at most $2\sqrt{2k}/(\sqrt{2k}+1) < 2$ times optimal by Theorem 4. The running time is

$$O\left(\frac{(\sqrt{2k})^{2k}(1 + 1/\sqrt{2k})^{2k}}{(\sqrt{2k} + 1)!}(\sqrt{2k} + 1)nm \log(n^2/m)\right) = O\left((2k)^k nm \log(n^2/m)\right)$$

which is polynomial for fixed k. The question remains open as to whether there is a polynomial-time approximation algorithm that works for arbitrary k and provides a constant guarantee, although Garg et al. [11] have devised a polynomial-time algorithm that works for arbitrary k and has worst-case ratio $O(\log k)$.

More recently, Erdös and Székely in [5], [6] proposed the following generalization of multiterminal cut. Suppose you are given a graph $G = (V, E)$ with weighted edges, and a partial k-coloring of the vertices, that is, a subset $V' \subseteq V$ and a function $f : V' \rightarrow \{1, 2, \ldots, k\}$. Can f be extended to a total function such that the total weight of edges that have different colored endpoints is minimized? The k-terminal cut problem is the special case where $|V'| = k$ and f is 1-1, that is, each color is initially assigned to precisely one vertex. It is easy to see that for general graphs, this problem is in fact equivalent to multiterminal cut. Simply merge all the vertices with the same color, call the resulting merged vertices "terminals," and find the minimum weight k-terminal cut for the resulting graph. For special classes of graphs, however, the "Colored Multiterminal Cut" problem can be more general. (The above merging trick need not for instance preserve planarity or acyclicity.) Nevertheless, in the case of trees, the dynamic programming algorithm for multiterminal cut mentioned in the Introduction extends in a natural way to the colored multiterminal cut problem, yielding an $O(nk)$ algorithm, as Erdös and Székely observe. This, is turn, implies that if G is such that deleting all the terminals renders it acyclic, then multiterminal cut can itself still be solved in $O(nk)$ time. (Simply split each terminal s_i into *degree*(s_i) separate vertices, one for each edge incident on s_i, assign color i to all the derived vertices, and apply the abovementioned algorithm for colored multiterminal cut on trees to the resulting graph [6]).

An obvious question is whether our algorithms for planar graphs also extend to Colored Multiterminal Cut problem. The answer is no. Colored multiterminal cut is clearly polynomial-time solvable if $k = 2$, even for general graphs. For any fixed $k \geq 3$, however, if remains NP-complete even for planar graphs and all weights equal to 1. The $k = 4$ case follows directly from our proof of Theorem 2. Simply use the four colors $x, \overline{x}, c^+$, and c^-, and assign x to each terminal x_i, $\overline{x}$ to each terminal $\overline{x}_i$, c to each terminal c_j^+, and c^- to each terminal c_j^-. It is straightforward to verify that the proof still goes through. For $k = 3$, the result can be proved by a transformation from PLANAR 3-COLORABILITY [9], [10], using a local replacement argument in which each edge is replaced by a partially colored structure designed to make it expensive for the endpoints of the original edge to get the same color. We leave the details to the enterprising reader. (Note that this last result provides us with an alternate proof of the NP-completeness of 3-TERMINAL CUT for general graphs: once again simply merge all vertices with the same color. The fact that such an operation may destroy planarity is in this case irrelevant.)

Returning to the original multiterminal cut problem, in our opinion the most interesting open problem is whether one can improve upon the approximation results of Theorem 4. Although polynomial-time algorithms with worst-case ratios arbitrarily close to 1 are unlikely in light of Theorem 5, can we do better than the $2(k-1)/k$ guarantee we proved for the Isolation Heuristic? Noga Alon (private communication, 1991) has observed that for the special cases of $k = 4$ and $k = 8$ improvements can be obtained using a variant of our approach. For $k = 4$, the Isolation Heuristic provides a guarantee of 3/2. An improved guarantee of 4/3 can be obtained as follows. For each partition of the terminals into sets S_1, S_2 of size two, use max flow techniques to compute the minimum cut that separates the terminals in S_1 from those in S_2. Output the union of the two best such cuts. The reader can readily verify that this union is a 4-terminal cut whose weight is at most 4/3 optimal. Note that this approach requires only three max flow computations versus the four needed by the Isolation Heuristic, so it is faster as well. (Cunningham reports in [3] that Zhang has independently obtained this $k = 4$ improvement.)

For $k = 8$, the guarantee of our theorem can be improved from 7/4 to 12/7. Here one computes minimum 2-terminal cuts on the basis of partitions of the set of terminals into sets of size four. It can be shown that the average weight of these cuts is at most 4/7 times the weight of an optimal 8-terminal cut, and that there exists a set of three of these cuts whose union is an 8-terminal cut and whose total weight is no more than average. This yields the claimed bound. Moreover, the running time is once again an improvement on the Isolation Heuristic, which in this case would require eight max flow computations. This is because we can show that it suffices to restrict attention to just seven of the 35 possible partitions of the eight terminals into sets of size four (the seven being derived from the rows of a Hadamard matrix).

Unfortunately, the above approach does not yield improvements over the Isolation Heuristic for any values of k other than 4 and 8. Is there some general technique that will improve on the Isolation Heuristic for arbitrarily large values of k? What about simply beating our bound for the case of $k = 3$?

Turning to our optimization algorithms for the planar case, the obvious question is whether the running times can be improved, although for the case of general k the improvement would have to be substantial to be interesting. For instance, although we would expect any algorithm to be exponential in k, the exponent containing k might not have to be attached to n. Could there be an algorithm whose running time was $c^k n^\alpha$, where α was independent of k?

Acknowledgment. The authors thank the anonymous referees for suggestions that improved the performance of our approximation algorithm for the multipair cut problem and more generally helped us clarify our presentation.

REFERENCES

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proceedings 33rd Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, Los Angeles, Calif., 1992, pp. 14–23.

[2] S. Chopra and M. R. Rao, *On the multiway cut polyhedron*, Networks 21 (1991), pp. 51–89.

[3] W. H. Cunningham, *The optimal multiterminal cut problem*, DIMACS Series in Disc. Math. and Theor. Comput. Sci. 5 (1991), pp. 105–120.

[4] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, *The complexity of multiway cuts*, extended abstract, 1983.

[5] P. L. Erdös and L. A. Székely, *Evolutionary trees: An integer multicommodity max-flow min-cut theorem*, Adv. in Appl. Math., to appear.

[6] ———, *Algorithms and min-max theorems for certain multiway cuts*, in Integer Programming and Combinatorial Optimization, E. Balas, G. Cornuéjols, and R. Kannan, eds., Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1992, pp. 334–345.

[7] L. R. Ford, Jr, and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[8] G. N. Frederickson, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[10] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems*, Theor. Comput. Sci., 2 (1976), pp. 237–267.

[11] N. Garg, V. V. Vazirani, and M. Yannakakis, *Approximate max-flow min-(multi)cut theorems and their applications*, in Proceedings of 25th Ann. ACM Symp. on Theory of Computing, Association for Computing Machinery, New York, 1993, pp. 698–707.

[12] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum-flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[13] O. Goldschmidt and D. S. Hochbaum, *A polynomial algorithm for the k-cut problem for fixed k*, Math. of O.R., 9 (1994), pp. 24–37.

[14] M. Grötschel, L. Lovász, and A. Schrijver, *The ellipsoid method and its consequences in combinatorial optimization*, Combinatorica, 1 (1981), pp. 169–198.

[15] X. He, *On the planar 3-cut problem*, J. Algorithms, 12 (1991), pp. 23–37.

[16] D. S. HOCHBAUM AND D. B. SHMOYS, *An $O(|V|^2)$ algorithm for the planar 3-cut problem*, SIAM J. Algebraic and Discrete Methods, 6 (1985), pp. 707–712.

[17] T. C. HU, *Integer Programming and Network Flows*, Addison- Wesley Publishing Co., Reading, MA 1969.

[18] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[19] D. LICHTENSTEIN, *Planar formulae and their uses*, SIAM J. Comput., 11 (1982), pp. 329–343.

[20] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[21] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Optimization, approximation, and complexity classes*, J. Comput. System Sci., 43 (1991), pp. 425–440.

[22] H. SARAN AND V. V. VAZIRANI, *Finding k-cuts within twice the optimal*, in Proceedings 32nd Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, Los Angeles, CA, 1991, pp. 743–751.

[23] H. S. STONE, *Multiprocessor scheduling with the aid of network flow algorithms*, IEEE Trans. Software Engrg., SE-3 (1977), pp. 85–93.

[24] M. YANNAKAKIS, P. C. KANELLAKIS, S. C. COSMADAKIS, AND C. H. PAPADIMITRIOU, *Cutting and partitioning a graph after a fixed pattern*, in Automata, Languages, and Programming, Lecture Notes in Comput. Sci., Vol. 154, Springer, Berlin, 1983, pp. 712–722.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 895–905, October 1994

DIAGNOSIS OF $t/(t+1)$-DIAGNOSABLE SYSTEMS*

A. DAS[†], K. THULASIRAMAN[‡], AND V. K. AGARWAL[§]

Abstract. A classic PMC (Preparata, Metze, and Chien) multiprocessor system [F. P. Preparata, G. Metze, and R. T. Chien, IEEE Trans. Electr. Comput., EC-16 (1967), pp. 848–854] composed of n units is said to be $t/(t+1)$ diagnosable [A. D. Friedman, *A new measure of digital system diagnosis*, in Dig. 1975 Int. Symp. Fault-Tolerant Comput., 1975, pp. 167–170] if, given a syndrome (complete collection of test results), the set of faulty units can be isolated to within a set of at most $t+1$ units, assuming that at most t units in the system are faulty. This paper presents a methodology for determining when a unit v can belong to an allowable fault set of cardinality at most t. Based on this methodology, for a given syndrome in a $t/(t+1)$-diagnosable system, the authors establish a necessary and sufficient condition for a vertex v to belong to an allowable fault set of cardinality at most t and certain properties of $t/(t+1)$-diagnosable systems. This condition leads to an $O(n^{3.5})t/(t+1)$-diagnosis algorithm. This $t/(t+1)$-diagnosis algorithm complements the $t/(t+1)$-diagnosability algorithm of Sullivan [*The complexity of system-level fault diagnosis and diagnosability*, Ph.D. thesis, Yale University, New Haven, CT, 1986].

Key words. fault diagnosis, fault isolation, PMC models, test assignment, graph algorithms, vertex cover sets

AMS subject classifications. 68M15, 68R10

1. Introduction.

Several models have been proposed in the literature for diagnosable system design. Of these, the now well-known PMC model introduced by Preparata, Metze, and Chien [1] has been extensively studied. In this model, each processor tests some of the other processors and produces test results, which are unreliable if the testing processor is itself faulty. The collection of all test results over the entire system is referred to as a *syndrome*. The classic constraint used in the study of diagnosable systems is to assume that the number of faulty processors in the entire system is upper-bounded by an integer t. A system is then said to be *t diagnosable* if given a syndrome, all processors can be correctly identified as faulty or fault free, provided that the number of faulty processors present in the system does not exceed t. Three problems of interest in this context are the *t-characterization problem* to determine the necessary and sufficient conditions for the system test assignment to be t diagnosable, the t-diagnosability problem to determine the largest value of t for which a given system is t diagnosable, and finally the t-diagnosis problem to locate the faulty units present in a t-diagnosable system, using a given syndrome.

Hakimi and Amin [2] gave a solution to the t-characterization problem. An $O(|E|n^{3/2})$ algorithm for the t-diagnosability problem was presented by Sullivan [3]. Subsequently, Raghavan [4] improved on this result by presenting an algorithm that runs in $O(nt^{2.5})$ time. Dahbura and Masson [5] published an $O(n^{2.5})$ algorithm for the t-diagnosis problem; a t-diagnosis algorithm with complexity $O(|E|+t^3)$ was presented by Sullivan [6].

The requirement that all the faulty processors in a multiprocessor system be identified exactly is rather restrictive. Friedman [7] introduced the concept of t/s-diagnosability. A multiprocessor system S is said to be t/s diagnosable if, given a syndrome, the set of faulty processors can be isolated to within a set of at most s processors provided that the number of faulty processors does not exceed t. Allowing some fault-free processors to be possibly identified as faulty permits the system to have far fewer tests. It has been shown that t/t-diagnosable systems with $n^*\lceil(t+1)/2\rceil$ tests can be constructed [8]. t/t-diagnosable systems have been studied extensively in the literature. Chwa and Hakimi [9] gave a characterization of t/t-diagnosable systems, Sullivan [10] presented a polynomial time algorithm

*Received by the editors September 4, 1990; accepted for publication (in revised form) April 18, 1993.

[†]Departement d'informatique et de recherche operationelle, Universitè de Montrèal, Montreal, Quebec, Canada.

[‡]Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada.

[§]Department of Electrical and Computer Engineering, McGill University, Montreal, Quebec, Canada.

for the t/t-diagnosability problem, and Yang, Masson, and Leonetti [11] presented an $O(n^{2.5})$ algorithm for the t/t-diagnosis problem. Sullivan also presented in [10] a polynomial time $t/(t+1)$-diagnosability algorithm based on a characterization of $t/(t+1)$-diagnosable systems also developed in [10].

The objective of this work is to develop an efficient diagnosis algorithm for $t/(t+1)$-diagnosable systems. The paper is organised as follows. In §2, we present certain basic definitions, notations, and results. With the objective of determining an effecient test for a vertex v to be in an allowable fault set of cardinality at most t, we then establish in §3 several properties of allowable fault sets and present a methodology for determining when a unit v belongs to an allowable fault set of cardinality at most t. Using these properties and certain properties of $t/(t+1)$-diagnosable systems, we develop in §4 a necessary and sufficient condition for a vertex v to belong to an allowable fault set of cardinality at most t. This leads to an $O(n^{3.5})$ algorithm for diagnosis of a $t/(t+1)$-diagnosable system. The work presented here is a revised version of our earlier paper [12].

2. Preliminaries. A multiprocessor system S consists of n units or processors, denoted by the set $U = \{u_1, u_2, \ldots, u_n\}$. Each unit is assigned a subset of other units for testing. Thus the test interconnection can be modeled as a directed graph $G = (U, E)$. The test outcome a_{ij}, which results when unit u_i tests unit u_j, has value 1 (respectively, 0) if u_i evaluates unit u_j to be faulty (respectively, fault free). Since all faults considered are permanent, the test outcome a_{ij} is reliable if and only if unit u_i is fault free. The collection of all test results over the entire system is referred to as a *syndrome*. If $a_{ij} = 0$ (respectively, 1) then u_i is said to have a 0-link (respectively, 1-link) to u_j and u_j is said to have a 0-link (respectively, 1-link) from u_i.

Given a syndrome, the *disagreement set* $\Delta_1(u_i)$ of $u_i \in U$ is defined as

$$\Delta_1(u_i) = \{u_j | a_{ij} = 1 \text{ or } a_{ji} = 1\}.$$

For a subset $W \subseteq U$,

$$\Delta_1(W) = \bigcup_{u_j \in W} \Delta_1(u_j).$$

Given a syndrome, the set of 0-*descendents* of u_i is represented by the set

$$D_0(u_i) = \{u_j : \text{ there is a directed path of 0-links from } u_i \text{ to } u_j\}$$

and for a set $W \subseteq U$, the 0-*ancestors* of W denote the set

$$A_0(W) = \{u_i : \exists u_j \in W \text{ such that } u_j \in D_0(u_i)\}.$$

For $u_i \in U$, $H_0(u_i)$ corresponds to the set $A_0(u_i) \cup \{u_i\}$.

DEFINITION 1 [5]. *Given a system* S *and a syndrome, a subset $F \subseteq U$ is an* allowable fault set (AFS) *if and only if*

1. *$u_i \in (U - F)$ and $a_{ij} = 0$ imply $u_j \in (U - F)$, and*
2. *$u_i \in (U - F)$ and $a_{ij} = 1$ imply $u_j \in F$.*

In other words, F is an AFS for a given syndrome if and only if the assumption that the units in F are faulty and the units in $U - F$ are fault free is consistent with the given syndrome. A *minimum allowable fault set* (MAFS) is an allowable fault set of minimum cardinality.

DEFINITION 2 [5]. *Given a system* S *and a syndrome, the* implied-fault set $L(u_i)$ *of $u_i \in U$ is the set of all units of* S *that may be deduced to be faulty under the assumption that u_i is fault free.*

It follows that

$$L(u_i) = \Delta_1(D_0(u_i)) \bigcup A_0(\Delta_1(D_0(u_i))).$$

Note that if $u_j \in L(u_i)$ then there exists a 1-link (u_k, u_l) or (u_l, u_k) such that there is a directed path of 0-links from u_i to u_k and a directed path of 0-links from u_j to u_l. This observation motivates the definition of an *implied-fault path* between u_i and u_j.

DEFINITION 3. *Given a system* S *and a syndrome, a path* P *between* u_i *and* u_j *is an* implied-fault path *if there exist units* u_k *and* u_l *on* P *such that the following are satisfied.*

(i) *All the links of* P *that lie between* u_i *and* u_k *constitute a directed path of* 0*-links from* u_i *to* u_k.

(ii) *All the links of* P *that lie between* u_j *and* u_l *constitute a directed path of* 0*-links from* u_j *to* u_l.

(iii) *Either* (u_k, u_l) *or* (u_l, u_k) *is a* 1*-link on* P.

Given two sets X_i and X_j, $X_i \oplus X_j$ denotes the *symmetric difference* of X_i and X_j. That is,

$$X_i \oplus X_j = (X_i - X_j) \bigcup (X_j - X_i).$$

If $u_i \in L(u_i)$ then clearly the unit u_i is faulty. Such a unit will be in every AFS. Without loss of generality, we assume in this paper that $u_i \notin L(u_i)$ for any $u_i \in U$.

The following lemmas determine a few properties of AFSs and implied faulty sets.

LEMMA 1 [5]. *Given a system* S *and a syndrome, each of the following statements holds.*

(1) *For* $u_i, u_j \in U$, $u_i \in L(u_j)$ *if and only if* $u_j \in L(u_i)$.

(2) *For* $u_i, u_j \in U$, *if* $a_{ij} = 0$ *then* $L(u_j) \subseteq L(u_i)$.

(3) *If* $F \subseteq U$ *is an* AFS, *then* $\bigcup_{u_i \in U-F} L(u_i) \subseteq F$.

LEMMA 2 [10]. *Given a system* S *and a syndrome, if* F_1 *and* F_2 *are* AFSs *then so is* $(F_1 \bigcup F_2)$.

LEMMA 3. *Given a system* S *and a syndrome, let* $F \subseteq U$ *be an* AFS *containing* $u_i \in U$. *Then* $H_0(u_i) \subseteq F$.

Proof. Suppose that $u_j \in H_0(u_i)$ is not a member of F. Since $u_j \in H_0(u_i)$ there exists a directed path of 0-links from u_j to u_i. Since $u_j \in U - F$ and $u_i \in F$, there exists an 0-link from $U - F$ to F on this path, contradicting the assumption that F is an AFS. □

For what follows, let $G' = (U', E')$ denote a general, undirected graph.

DEFINITION 4. *A subset* $K \subseteq U'$ *is called a* vertex cover set (VCS) [13] *of* G' *if every edge in* G' *is incident on at least one vertex in* K. *A* minimum vertex cover set (MVCS) *is a* VCS *of minimum cardinality in* G'.

DEFINITION 5. *A subset* $M \subseteq E'$ *is called a* matching [13] *if no vertex in* U' *is incident on more than one edge in* M. *A* maximum matching *is a matching of maximum cardinality in* G'.

A *bipartite graph*, with bipartition (X, Y), is one whose vertex set can be partitioned into two subsets X and Y such that every edge is incident to a vertex in X and a vertex in Y. Finally, for $u_i \in U'$, $N(u_i)$ denotes the set of all vertices that are adjacent to u_i.

3. Basic properties of allowable fault sets.

In this section we establish certain properties of allowable fault sets with respect to a given syndrome. Our study is directed toward investigating conditions for a vertex v to be in an allowable fault set of cardinality at most t. For this purpose we use the notion of implied-fault set and the implied-fault graph used by Dahbura and Masson [5] in their study.

Given a syndrome for a system S, define the implied-fault graph $G^* = (U^*, E^*)$ to be an undirected graph such that $U^* = U$ and $E^* = \{(u_i, u_j) : u_i \in L(u_j)\}$. For $u \in U$, let G_u^*

denote the subgraph of G^* obtained after all units in $H_0(u)$ and all edges incident on these units have been removed from G^*. Let K_u represent an MVCS of G_u^* and let $G - H_0(u)$ denote the subgraph of G where all vertices in $H_0(u)$ along with all edges incident on these vertices have been removed from G. Finally, we define $G^*(F)$ to be the subgraph of G^* such that all edges that connect vertices entirely inside F have been deleted.

Recall that we have assumed that $u_i \notin L(u_i)$ for any $u_i \in U$. This means that G^* has no self-loops.

The results of the following lemma can be found in [5]. We present this lemma for the sake of completeness.

LEMMA 4. *Given a syndrome for a system* S, *we have the following.*

(i) *Every* AFS *of G is a* VCS *of G^*.*

(ii) *If $F \subseteq U$ is a minimal* VCS *of G^*, then F is an* AFS *of G.*

(iii) *$F \subseteq U$ is an* MAFS *of G if and only if F is an* MVCS *of G^*.*

Proof. (i) Let F be an AFS of G for the given syndrome. Assume F is not a VCS of G^*. Then there exist $u_i, u_j \in U - F$ such that (u_i, u_j) is an edge in G^*. Since all edges from $U - F$ into F in G are 1-links (F is an AFS of G) and an implied-fault path between u_i and u_j can contain only one 1-link, all vertices that lie on an implied-fault path between u_i and u_j must belong to $U - F$. But this implies that there is a 1-link between two vertices in $U - F$, contradicting the assumption that F is an AFS of G. This shows that (i) holds.

(ii) Let F be a minimal VCS of G^*. Assume (ii) does not hold. Then at least one of the following conditions is satisfied.

(a) There exist $u_i, u_j \in U - F$ with $a_{ij} = 1$.

(b) There exist $u_j \in F, u_i \in U - F$ with $a_{ij} = 0$.

Assume (a) holds. Then the edge (u_i, u_j) is in G^*. But this contradicts the fact that F is a VCS of G^* since neither u_i nor u_j is a member of F.

Now assume (b) holds and (a) does not hold. Since F is a minimal VCS of G^* there exists a unit u_k in $U - F$ such that (u_j, u_k) is an edge in G^*; otherwise $F - \{u_j\}$ will be a VCS, contradicting the minimality of F. Hence $u_j \in L(u_k)$. Since $a_{ij} = 0$, it follows that $u_i \in L(u_k)$ and so (u_i, u_k) is an edge in G^*. Since neither u_i nor u_k is a member of F, this contradicts the fact that F is a VCS of G^*.

(iii) Statement (iii) follows from (i) and (ii). □

LEMMA 5. *F is an* AFS *in G of minimum cardinality containing unit v if and only if $H = F - H_0(v)$ is an* MAFS *of $G - H_0(v)$.*

Proof. We first prove necessity. We first show that H is an AFS of $G - H_0(v)$. Since $U - F = (U - H_0(v)) - H$ and F is an AFS of G, all edges within $(U - H_0(v)) - H$ are 0-links and all edges from $(U - H_0(v)) - H$ into H are 1-links. Hence H is an AFS of $G - H_0(v)$. To show that H is an MAFS of $G - H_0(v)$, assume H_1 is an AFS of $G - H_0(v)$. Clearly all edges with both vertices incident on vertices in $(U - H_0(v)) - H_1$ are 0-links and all edges from $U - H_0(v) - H_1$ into H_1 are 1-links. Now consider edges from $(U - H_0(v)) - H_l$ into $H_0(v)$. These edges must all be 1-links, otherwise the vertices incident on these edges would all belong to $H_0(v)$. This shows that the set $H_1 \bigcup H_0(v)$ is an AFS of G. Hence if $|H_1| < |H|$ then $H_1 \bigcup H_0(v)$ is an AFS of smaller cardinality than F, contradicting the fact that F is an AFS of minimum cardinality containing v. Hence $|H_1| \geq |H|$ and H is an MAFS of $G - H_0(v)$.

We will now prove sufficiency. If $F - H_0(v)$ is an MAFS of $G - H_0(v)$ then, as we have shown in the proof of necessity, F is an AFS of G. If F is not an AFS in G of minimum cardinality containing v, then let F_1 be an AFS of G containing v with $|F_1| < |F|$. But then $F_1 - H_0(v)$, from the necessity part, would be an AFS of $G - H_0(v)$ of smaller cardinality than $F - H_0(v)$, which is a contradiction. □

LEMMA 6. *For $v \in U$, $(G - H_0(v))^* = G_v^*$.*

Proof. Since the vertex sets of both graphs are the same, we need only show that the edge sets are identical. Clearly every edge in $(G - H_0(v))^*$ is in G_v^*. Now assume that there is an edge (u_i, u_j) in G_v^* that is not in $(G - H_0(v))^*$. Then every implied-fault path in G between u_i and u_j must contain at least one vertex from $H_0(v)$. But this implies that either u_i or u_j is a member of $H_0(v)$, contradicting the assumption that both vertices are members of $G - H_0(v)$. Hence the two edge sets are also identical. □

LEMMA 7. *Given a syndrome for a system* S, *let* $F \subseteq U$ *be an* AFS *containing* $v \in U$. *Then* $F - H_0(v)$ *is a* VCS *of* G_v^*.

Proof. Let $F_1 = F - H_0(v)$. From Lemma 3 and the proof of Lemma 5, it follows that F_1 is an AFS of $G - H_0(v)$. Then from Lemma 4, F_1 is a VCS of $(G - H_0(v))^*$. Thus, by Lemma 6, F_1 is a FCS of G_v^*. □

THEOREM 1. *Given a syndrome for a system* S, F *is an* AFS *of minimum cardinality among all allowable fault sets that contain unit* $v \in U$ *if and only if* $F - H_0(v)$ *is a* MVCS *of* G_v^*.

Proof. The proof follows from Lemmas 4, 5 and 6. □

The condition given in the above theorem can be used to test whether a unit belongs to an AFS of cardinality at most t for a given syndrome. However, this condition requires determining an MVCS for a general undirected graph, a problem known to be NP complete. Therefore we would like to develop a test that requires determining an MVCS of a bipartite graph. With this objective in mind, we first define a bipartite graph for each vertex v. This bipartite graph is derived from G_v^*. We then relate an MVCS of this graph to an AFS containing vertex v and establish certain properties of this AFS that will be used in the following section to develop the appropriate diagnosis algorithm.

Given a system S and a syndrome, define $B = (U_B, E_B)$ to be the undirected bipartite graph with bipartition (X, Y) where

$$X = \{x_1, \dots, x_n\}, \qquad Y = \{y_1, \dots, y_n\}$$

and

$$E_B = \{(x_i, y_j) : u_i \in L(u_j) \text{ in S}\}.$$

For $v \in U$, define the undirected bipartite graph $B_v = (U_v, E_v)$ with bipartition (X_v, Y_v) to be the vertex-induced subgraph of B such that

$$X_v = \{x_i : u_i \in U - H_0(v)\}, \qquad Y_v = \{y_i : u_i \in U - H_0(v)\}.$$

For each vertex v in G, let

$$t_v = t - |H_0(v)|$$

and

$$U_v = U - H_0(v).$$

THEOREM 2. *Given a syndrome for a system* S, *a unit* $v \in U$ *does not belong to any* AFS *of cardinality at most* t *if* B_v *has an* MVCS *of cardinality greater than* $2t_v$.

Proof. Let the cardinality of an MVCS of B_v be greater than $2t_v$. Assume $v \in U$ belongs to an AFS F such that $|F| \leq t$. Let $H = F - H_0(v)$. Since, by Lemma 3, $H_0(v) \subseteq F$, it follows that $|H| = |F| - |H_0(v)| \leq t - |H_0(v)| = t_v$. Define $B_X(H) = (U_X, E_X)$ to be the vertex-induced subgraph of B_v, where

$$U_X = \{x_i : u_i \in H\} \cup \{y_i : u_i \in U - H\}.$$

$B_Y(H) = (U_Y, E_Y)$ is defined to be the vertex-induced subgraph of B_v, where

$$U_Y = \{x_i : u_i \in U - H\} \cup \{y_i : u_i \in H\}.$$

Clearly $F_X = \{x_i : u_i \in H\}$ and $F_Y = \{y_i : u_i \in H\}$ are VCSs of $B_X(H)$ and $B_Y(H)$, respectively. It follows that $F_B = F_X \cup F_Y$ is a VCS of $B_X(H) \cup B_Y(H)$. Since F is an AFS, in G^* there are no edges connecting vertices of $U - F$. From this it follows that every edge in $B_v - B_X(H) - B_Y(H)$ connects vertices in F_B. Therefore F_B is a VCS of B_v, contradicting our assumption that the cardinality of an MVCS of B_v is greater than t_v. Hence v is not contained in any AFS of cardinality at most t. □

In the following we use $F_B(v)$ to denote an MVCS of B_v. For a given $F_B(v)$ let

$$F_I = \{u_i | x_i \in F_B(v) \text{ and } y_i \in F_B(v)\}$$

and

$$F_v = \{u_i | x_i \in F_B(v) \text{ or } y_i \in F_B(v)\}.$$

We now proceed to establish certain properties of F_v.

LEMMA 8. *$F_v \cup H_0(v)$ is an* AFS *of G.*

Proof. Assume the contrary. Then at least one of the following conditions is satisfied.

(a) There exist $u_i, u_j \in U - F_v - H_0(v)$ with $a_{ij} = 1$.

(b) There exist $u_j \in F_v \cup H_0(v)$ and $u_i \in U - (F_v \cup H_0(v))$ with $a_{ij} = 0$.

Assume (a) holds. Then the edge (u_i, u_j) is in G_v^*. Hence (x_i, y_j) is an edge in B_v. But this contradicts the fact that $F_B(v)$ is a VCS of B_v since neither x_i nor y_j is a member of $F_B(v)$.

Now assume (b) holds and (a) does not hold. Clearly $u_j \notin H_0(v)$; otherwise u_i would also belong to $H_0(v)$. Thus $u_j \in F_v$. Hence either x_j or y_j is a member of $F_B(v)$. Without loss of generality, let $x_j \in F_B(v)$. Since $F_B(v)$ is an MVCS of B_v, there exists y_k in B_v with $y_k \notin F_B(v)$ such that (x_j, y_k) is an edge in B_v. Hence $u_j \in L(u_k)$. Since $a_{ij} = 0$, $u_i \in L(u_k)$. Hence (x_i, y_k) is an edge in B_v. Since neither x_i nor y_k is a member of $F_B(v)$, this contradicts the fact that $F_B(v)$ is a VCS of B_v. □

LEMMA 9. *Given a syndrome for a system* S, *a unit $v \in U$, and an* MVCS *$F_B(v)$ of B_v, we have the following.*

(i) *In G^*, there is no edge (u_i, u_j) with $u_i \in U - (F_v \cup H_0(v))$ and $u_j \in F_v - F_I$.*

(ii) *In G, there is no edge (u_i, u_j) with $u_i \in U - (F_v \cup H_0(v))$ and $u_j \in F_v - F_I$.*

Proof. (i) Assume the contrary. Let (u_i, u_j) be an edge from $U - (F_v \cup H_0(v))$ into $F_v - F_I$ in G^*. Then either x_j or y_j is not a member of $F_B(v)$. Thus in B_v either the edge (x_i, y_j) or the edge (x_j, y_i) is not incident on any vertex in $F_B(v)$, contradicting the fact that $F_B(v)$ is a VCS of B_v.

(ii) By Lemma 8, the set $F_v \cup H_0(v)$ is an AFS of G. Thus every edge from $U - (F_v \cup H_0(v))$ into $F_v - F_I$ in G must be a 1-link. So if such an edge (u_i, u_j) exists in G, then (u_i, u_j) is an edge in G^*. Thus from (i) it follows that there is no edge (u_i, u_j) in G with $u_i \in U - (F_v \cup H_0(v))$ and $u_j \in F_v - F_I$. □

LEMMA 10. *Every* AFS *of G contained in $F_v \cup H_0(v)$ contains the subset F_I.*

Proof. To show that every AFS of G contained in $F_v \cup H_0(v)$ contains the subset F_I it suffices to show that every VCS of G^* contained in $F_v \cup H_0(v)$ contains F_I. The above assertion holds if every vertex in F_I is incident on some vertex of $U - (F_v \cup H_0(v))$ in G^*. Assume the contrary. Let u_k be a vertex in F_I that is not incident on any vertex of the set $U - (F_v \cup H_0(v))$. Then let $W_B(v) = \{x_i | u_i \in F_v\} \cup \{y_i | u_i \in F_I - \{u_k\}\}$. From Lemma 9(i) and the construction of B_v it follows that in B_v, there is no edge (x_i, y_j) with $u_i \in U - (F_v \cup H_0(v))$ and $u_j \in F_v - F_I$. So $W_B(v)$ is a VCS of B_v. But $|W_B(v)| = |F_B(v)| - 1$. This contradicts the assumption that $F_B(v)$ is an MVCS of B_v. Thus every vertex in F_I is incident on some vertex of $U - (F_v \cup H_0(v))$ in G^*. This implies that every VCS of G^* contained in $F_v \cup H_0(v)$ contains the subset F_I. By Lemma 4, it follows that every AFS of G contained in $F_v \cup H_0(v)$ contains the subset F_I. □

4. $O(n^{3.5})$ algorithm for diagnosis of a $t/(t+1)$-diagnosable system. In this section we present a necessary condition for a system S to be $t/(t+1)$ diagnosable. We then establish that in the case of a $t/(t+1)$-diagnosable system, the condition of Theorem 2 is both necessary and sufficient for a vertex v to be in an AFS of cardinality at most t. This will lead to an $O(n^{3.5})$ diagnosis algorithm to isolate all faulty units to within at most $t+1$ units in a $t/(t+1)$-diagnosable system.

Recall (§1) that a system S is said to be $t/(t+1)$ diagnosable if, given a syndrome, the set of faulty processors can be isolated to within a set of at most $t+1$ processors provided that the number of faulty processors does not exceed t.

It should be observed that a system is trivially $t/(t+1)$ diagnosable if $n = t+1$. Thus it is required that $0 \le t < n-1$. It should be noted [1] that under these conditions $n \ge 2t+1$ for $t/(t+1)$-diagnosable systems.

THEOREM 3. *If* S, *a multiprocessor system with test interconnection* $G = (U, E)$, *is* $t/(t+1)$ *diagnosable then for all* $X_i, X_j \subseteq U$ *with* $|X_i| > t$, $X_j \not\subseteq X_i$, *and* $|X_i| + |X_j| \le 2t$, *there exists a test from* $U - X_i - X_j$ *into* $X_i \oplus X_j$.

Proof. Assume S is $t/(t+1)$ diagnosable but the condition does not hold. Then there exist $X_i, X_j \subseteq U$ with $|X_i| > t$, $X_j \not\subseteq X_i$, $|X_i| + |X_j| \le 2t$ such that there is no test from $U - X_i - X_j$ into $X_i \oplus X_j$.

Since $|X_i| > t$ and $X_j \not\subseteq X_i$, $|X_i \cup X_j| > t+1$, we construct two sets F_1 and F_2 from X_i and X_j by moving elements from $X_i - X_j$ into $X_j - X_i$ until F_1 and F_2 have cardinality at most t. Thus we obtain two sets F_1 and F_2 with $|F_1| \le t$, $|F_2| \le t$ such that there is no test from $U - F_1 - F_2$ into $F_1 \oplus F_2$. Now consider the following syndrome (see Fig. 1) where for each edge $(u_k, u_l) \in E$ the outcome is defined as follows.

Case 1. $u_l \in U - (F_1 \cup F_2)$; then set $a_{kl} = 0$.
Case 2. $u_l \in F_1 \cup F_2$.
2.1. $u_l \in F_1 \cap F_2$; then set $a_{kl} = 1$.
2.2. $u_k, u_l \in F_1 - F_2$; then set $a_{kl} = 0$.
2.3. $u_k, u_l \in F_2 - F_1$; then set $a_{kl} = 0$.
2.4. $u_k \in F_1 \cap F_2$; then set $a_{kl} = 1$.
2.5. $u_k \in F_1 - F_2$ and $u_l \in F_2 - F_1$; then set $a_{kl} = 1$.
2.6. $u_k \in F_2 - F_1$ and $u_l \in F_1 - F_2$; then set $a_{kl} = 1$.

Both F_1 and F_2 are allowable fault sets of cardinality at most t for the given syndrome and $|F_1 \cup F_2| > t+1$. This contradicts the assumption that S is $t/(t+1)$ diagnosable. □

Recall from the previous section that $F_B(v)$ is an MVCS of B_v and F_v and F_I are sets derived from $F_B(v)$.

THEOREM 4. *Given a syndrome for a* $t/(t+1)$*-diagnosable system* S, *a unit* $v \in U$ *belongs to an* AFS *of cardinality at most t if and only if* $|F_B(v)| \le 2t_v$.

Proof. If $|F_B(v)| > 2t_v$ then, by Theorem 2, G does not contain an AFS of cardinality at most t containing the unit v.

Now assume $|F_B(v)| \le 2t_v$. If $F_v \cup H_0(v)$ contains an AFS of G of cardinality at most t containing the unit v then we are through. So assume $|F_B(v)| \le 2t_v$ and $F_v \cup H_0(v)$ does not contain any AFS of G of cardinality at most t containing the unit v. From Lemma 8, $F_v \cup H_0(v)$ is an AFS of G containing the unit v. If $F_v = F_I$ then $|F_v \cup H_0(v)| \le t$ since $|F_B(v)| \le 2t_v$. So we further assume that $F_v \ne F_I$. Since G^* does not contain any units with

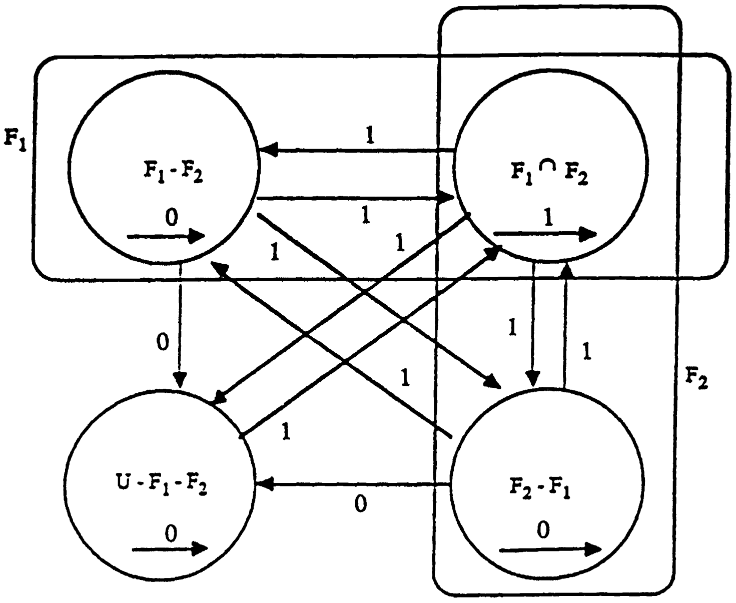


FIG. 1. *Two allowable fault sets generating a common syndrome.*

self-loops and $F_v \neq F_I$, the subset $|F_v - F_I| \geq 2$. Let F_α be an AFS of smallest cardinality containing unit v such that $F_\alpha \subseteq F_v \cup H_0(v)$. Clearly $|F_\alpha| > t$.

By Lemma 10 every AFS of G contained in $F_v \cup H_0(v)$ contains the subset F_I, and since $v \in F_\alpha$, it follows that $F_I \cup H_0(v) \subseteq F_\alpha$.

We next show that $F_\alpha \neq F_v \cup H_0(v)$. Let $u_i \in F_v - F_I$. Then $W = F_v - \{u_i\}$ is a VCS of G_v^* because, by Lemma 9(i), in G_v^* there is no edge (u_i, u_j) with $u_i \in U - (F_v \cup H_0(v))$ and $u_j \in F_v - F_I$. Hence by Lemma 6, W is a VCS of $(G - H_0(v))^*$. This means that, by Lemma 4(ii), W contains an AFS of $G - H_0(v)$. Thus $W \cup H_0(v)$ has an AFS of G containing unit v and of cardinality less than that of $F_v \cup H_0(v)$. Now let $F_a = F_\alpha - (H_0(v) \cup F_I)$, $F_b = (F_v \cup H_0(v)) - F_\alpha)$, and $F_\beta = F_b \cup H_0(v) \cup F_I$. Since F_α is an AFS of smallest cardinality containing v such that $F_\alpha \subseteq F_v \cup H_0(v)$, it follows that $|F_b| = |(F_v \cup H_0(v)) - F_\alpha| > 0$ and $F_\beta \not\subseteq F_\alpha$ (see Fig. 2).

Now

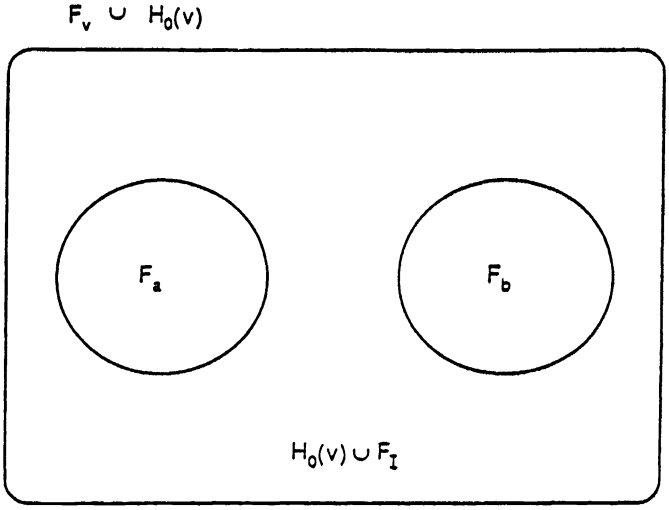
$$\begin{aligned} |F_\alpha| + |F_\beta| &= 2|F_I| + 2|H_0(v)| + |F_a| + F_b| \\ &= |F_B(v)| + 2|H_0(v)| \\ &\leq 2t_v + 2|H_0(v)| \leq 2t \end{aligned}$$

(see also Fig. 2).

Since $U - F_\alpha - F_\beta = U - (F_v \cup H_0(v))$ and $F_\beta \oplus F_\alpha = F_v - F_I$, it follows from Lemma 9(ii) that in G there is no edge (u_i, u_j) with $u_i \in U - (F_v \cup H_o(v)$ and $u_j \in F_v - F_I$.

Thus we have $|F_\alpha| > t$, $F_\beta \not\subseteq F_\alpha$, $|F_\alpha| + |F_\beta| \leq 2t$, and there is no test from $U - F_\alpha - F_\beta$ into $F_\alpha \oplus F_\beta$. This, by Theorem 3, contradicts our assumption that the system S is $t/(t+1)$ diagnosable. □

Given a valid syndrome for a $t/(t+1)$-diagnosable system S and a unit v in S, we have shown that the bipartite graph B_v has an MVCS of cardinality at most $2t_v$ if and only if G has an AFS of cardinality at most t containing the unit v. Since an MVCS of a bipartite graph has



$$F_\alpha = F_a \cup H_0(v) \cup F_I$$

$$F_\beta = F_b \cup H_0(v) \cup F_I$$

$$F_v = F_a \cup F_b \cup F_I$$

$$|F_{B(v)}| = 2|F_I| + |F_a| + |F_b|$$

$$F_\alpha \cap F_\beta = H_0(v) \cup F_I$$

$$F_\alpha \ominus F_\beta = F_a \cup F_b$$

FIG. 2. *Illustration for proof of Theorem* 4.

the same cardinality as a maximum matching of the bipartite graph [13], it follows that B_v has a maximum matching of cardinality at most $2t_v$ if and only if G has an AFS of cardinality at most t containing the unit v. Thus Theorem 4 can be stated in an equivalent manner as follows.

THEOREM 5. *Given a syndrome for a $t/(t+1)$-diagnosable system* S, *a unit $v \in U$ belongs to an* AFS *of cardinality at most t if and only if B_v has a maximum matching of cardinality at most* $2t_v$.

The above theorem suggests the following $t/(t+1)$-diagnosis algorithm.

```
ALGORITHM. Diagnosis of a $t/(t+1)$-diagnosable system
Step 1. Given a $t/(t+1)$-diagnosable system S and a valid syndrome, construct the
        bipartite graph $B = (U_B, E_B)$ with bipartition $(X, Y)$.
Step 2. Set $F = \phi$; for all $v \in U$, label $v$ unmarked.
Step 3. While there exists an unmarked $v \in U$
          begin
            3.1. Label $v$ marked.
            3.2. Set $t_v = t - |H_0(v)|$.
            3.3. Construct $B_v$ from $B$.
            3.4. Compute a maximum matching $K_v$ of $B_v$ using the Hopcroft/Karp al-
                 gorithm [14].
            3.5. If $|K_v| \leq 2t_v$ then add $v$ to $F$.
          end
Step 4. $F$ is the required set.
```

The proof of correctness of the above algorithm is as follows.

Essentially the algorithm proceeds as follows. Given a syndrome, for each unit $v \in U$ the algorithm tests if the cardinality of a maximum matching of B_v is at most $2t_v$. If unit v satisfies this requirement, then v is added to the set F. When the algorithm terminates we have

$$F = \{v : v \in U \text{ and } B_v \text{ has a maximum matching of cardinality at most } 2t_v\}.$$

Given a valid syndrome, in $t/(t+1)$-diagnosis we are required to isolate all faulty units to within a set of cardinality at most $t+1$. In other words, we need to determine the set of all units that are likely to be faulty under the given syndrome. By the definition of a $t/(t+1)$-diagnosable system, a unit v is likely to be faulty if and only if it belongs to an AFS of cardinality at most t. It then follows from Theorem 5 that a unit v is likely to be faulty if and only if B_v has a maximum matching of cardinality at most $2t_v$. The set F determined by the algorithm is therefore the required set consisting of all units that are likely to be faulty. By the definition of a $t/(t+1)$-diagnosable system, this set has cardinality at most $t+1$. Thus F is the required set isolating all the faulty units to within a set of cardinality at most $t+1$. This completes the proof of correctness of our $t/(t+1)$-diagnosis algorithm.

The bipartite graph in step 1 can be constructed in $O(n^{2.5})$ operations [5]. Step 2 requires $O(n)$ operations. The computation within step 3 is dominated by the computation of a maximum matching that requires $O(n^{2.5})$ operations [14]. Since step 3 is performed for each unit in U, the complexity of the entire algorithm is $O(n^{3.5})$.

5. Conclusions. In this paper we have studied the problem of diagnosing $t/(t+1)$-diagnosable systems. We presented a diagnosis for $t/(t+1)$-diagnosable systems that runs in $O(n^{3.5})$ time. This algorithm is based on the structure of allowable fault sets (§3) and on certain properties of $t/(t+1)$-diagnosable systems (§4).

REFERENCES

[1] F. P. Preparata, G. Metze, and R. T. Chien, *On the connection assignment problem of diagnosable systems*, IEEE Trans. Electr. Comput., EC-16 (1967), pp. 848–854.

[2] S. L. Hakimi and A. Amin, *Characterization of the connection assignment of diagnosable systems*, IEEE Trans. Comput., C-23 (1974), pp. 86–88.

[3] G. F. Sullivan, *A polynomial time algorithm for fault diagnosability*, in Proc. 25th Annual Symp. Foundations Comput. Sci., Orlando, FL, (1984), pp. 148–156.

[4] V. Raghavan, *Diagnosability issues in multiprocessor systems*, Ph.D. thesis, University of Minnesota, Minneapolis, MN, 1989.

[5] A. T. Dahbura and G. M. Masson, *A practical variation of the $O(n^{2.5})$ fault diagnosis algorithm*, in 14th Int. Symp. Fault-Tolerant Comput., 1984, pp. 428–433.

[6] G. F. Sullivan, *An $O(t^3 + |E|)$ fault identification algorithm for diagnosable systems*, IEEE Trans. Comput., Pans, C-37 (1988), pp. 388–397.

[7] A. D. Friedman, *A new measure of digital system diagnosis*, in Dig. 1975 Int. Symp. Fault-Tolerant Comput., (1975), pp. 167–170.

[8] A. Kavianpour and A. D. Friedman, *Efficient design of easily diagnosable systems*, in Proc. 3rd USA-Japan Comput. Conf., 1978, pp. 251–257.

[9] K. Y. Chwa and S. L. Hakimi, *On fault identification in diagnosable systems*, IEEE Trans. Comput., C-30 (1981), pp. 414–422.

[10] G. Sullivan, *The complexity of system-level fault diagnosis and diagnosability*, Ph.D. thesis, Yale University, New Haven, CT, 1986.

[11] C. L. Yang, G. M. Masson, and R. A. Leonetti, *On fault identification and isolation in t_1/t_1-diagnosable systems*, IEEE Trans. Comput., C-35 (1986), pp. 639–643.

[12] A. Das, K. Thulasiraman, V. K. Agarwal, and K. B. Lakshmanan, *t/s-diagnosable systems: A characterization and diagnosis algorithm*, in Proc. 15th International Conference on Graph-Theoretic Concepts in Computer Science, Rolduc, Holland, 1989, pp. 34–45.

[13] J. A. BONDY AND U. S. R. MURTHY, *Graph Theory with Applications*, Elsevier North-Holland, Amsterdam, 1976.

[14] J. E. HOPCROFT AND R. M. KARP, *A $n^{2.5}$ algorithm for maximum matching in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 906–933, October 1994

IMPROVED ALGORITHMS FOR BIPARTITE NETWORK FLOW*

RAVINDRA K. AHUJA†, JAMES B. ORLIN‡, CLIFFORD STEIN§, AND ROBERT E. TARJAN¶

Abstract. In this paper, network flow algorithms for bipartite networks are studied. A network $G = (V, E)$ is called *bipartite* if its vertex set V can be partitioned into two subsets V_1 and V_2 such that all edges have one endpoint in V_1 and the other in V_2. Let $n = |V|$, $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$ and assume without loss of generality that $n_1 \leq n_2$. A bipartite network is called *unbalanced* if $n_1 \ll n_2$ and *balanced* otherwise. (This notion is necessarily imprecise.) It is shown that several maximum flow algorithms can be substantially sped up when applied to unbalanced networks. The basic idea in these improvements is a *two-edge push rule* that allows one to "charge" most computation to vertices in V_1, and hence develop algorithms whose running times depend on n_1 rather than n. For example, it is shown that the two-edge push version of Goldberg and Tarjan's FIFO preflow-push algorithm runs in $O(n_1 m + n_1^3)$ time and that the analogous version of Ahuja and Orlin's excess scaling algorithm runs in $O(n_1 m + n_1^2 \log U)$ time, where U is the largest edge capacity. These ideas are also extended to dynamic tree implementations, parametric maximum flows, and minimum-cost flows.

Key words. network flow, bipartite graphs, maximum flow, minimum-cost flow, parametric maximum flow, parallel algorithms

AMS subject classifications. 90B10 , 68Q25 , 68R10

1. Introduction.

In this paper, we study network flow algorithms for bipartite networks. A network $G = (V, E)$ is called *bipartite* if its vertex set V can be partitioned into two subsets V_1 and V_2 such that all edges have one endpoint in V_1 and the other in V_2. Let $n = |V|$, $n_1 = |V_1|$, $n_2 = |V_2|$, $m = |E|$, and assume without loss of generality that $n_1 \leq n_2$. We call a bipartite network *unbalanced* if $n_1 \ll n_2$ and *balanced* otherwise. We show that several maximum flow algorithms can be substantially sped up when applied to *unbalanced networks*. At first glance, it may appear that unbalanced networks are of limited practical utility. This is not true, however. Gusfield, Martel, and Fernandez-Baca [21] have compiled a list of many practical applications of unbalanced networks. Further applications of unbalanced networks appear in [14].

Specialized bipartite flow algorithms for unbalanced networks were first studied by Gusfield, Martel, and Fernandez-Baca [21]. They developed modifications of the algorithms of Karzanov [25] and Malhotra, Pramodh Kumar, and Maheshwari (MPM)[27] for the maximum flow problem that improved their running times from $O(n^3)$ to $O(n_1^2 n_2)$. For the bounded degree case, i.e., when the degree of each vertex in V_2 is bounded by a fixed constant, they

*Received by the editors May 20, 1991; accepted for publication (in revised form) June 4, 1993.

†Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur 208016, India. This author's research was supported in part by Presidential Young Investigator grant 8451517-ECS of the National Science Foundation, by grant AFOSR-88-0088 from the Air Force Office of Scientific Research, and by grants from Analog Devices, Apple Computers, Inc., and Prime Computer.

‡Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Research partially supported by Presidential Young Investigator grant 8451517-ECS of the National Science Foundation, by grant AFOSR-88-0088 from the Air Force Office of Scientific Research, and by grants from Analog Devices, Apple Computers, Inc., and Prime Computer.

§Department of Computer Science, Dartmouth College, Hanover, New Hampshire 03755. Some of the results in this paper were part of this author's undergraduate thesis at Princeton University. Some of the work was done while this author was a graduate student at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Research partially supported by a graduate fellowship from AT&T. Additional support provided by Air Force contract AFOSR-86-0078 and by a National Science Foundation PYI grant awarded to David Shmoys, with matching funds from IBM, Sun Microsystems, and the United Parcel Service.

¶Department of Computer Science, Princeton University, Princeton, New Jersey 08544, and NEC Research Institute, Princeton, New Jersey 08540. Research at Princeton University partially supported by the National Science Foundation grant DCR-8605952, and the Office of Naval Research contract N00014-91-K-1463.

developed a further modification of the MPM algorithm that runs in $O(n_1 m + n_1^3)$ time. We suggest several algorithms for the maximum flow problem on unbalanced networks that improve the running times of Gusfield et al. for all classes of unbalanced networks.

Gusfield [20] has shown that on a particular bipartite network in which each vertex in V_2 has constant degree, an algorithm similar to the FIFO preflow-push maximum flow algorithm of Goldberg and Tarjan [15],[16] runs in $O(n_1 m + n_1^3)$ time. Further, he observes that this result extends to parametric maximum flow; he solves a series of n_1 maximum flow problems in $O(n_1 m + n_1^3)$ time. We have similar results, which were obtained independently and apply to a more general class of networks.

We begin with the observation of Gusfield, Martel, and Fernandez-Baca [21] that the time bounds for several maximum flow algorithms automatically improve when the algorithms are applied *without modification* to unbalanced networks. A careful analysis of the running times of these algorithms reveals that the worst-case bounds depend on the number of edges in the longest vertex-simple path in the network. We call this the *path length* of the network and denote it by L. For a general network, L may be as large as $n - 1$; but, for a bipartite network, L is at most $2n_1 + 1$. Hence for unbalanced networks the path length is much less than n, and we get an automatic improvement in running times. As an example, consider Dinic's algorithm [10] for the maximum flow problem. This algorithm constructs $O(L)$ layered networks and finds a blocking flow in each one. Each blocking flow computation performs $O(m)$ augmentations and each augmentation takes $O(L)$ time. Consequently, the running time of Dinic's algorithm is $O(L^2 m)$. Thus, when applied to unbalanced networks, the running time of Dinic's algorithm improves from $O(n^2 m)$ to $O(n_1^2 m)$. Column 3 of Table 1.1 summarizes these improvements for several network flow algorithms.

We obtain further running-time improvements by modifying the algorithms. This modification applies only to preflow-push algorithms [2], [3], [14]–[17]; we call it the *two-edge push rule*. According to this rule, we always push flow from a vertex in V_1 and push flow on two edges at a time, in a step called a *bipush*, so that no excess accumulates at vertices in V_2. This rule allows us to charge all computations to examinations of vertices in V_1, though without this rule they might be charged to vertices in V_2. As an outcome of this rule, we develop algorithms whose running times depend on n_1 rather than n. We incorporate the two-edge push rule in several maximum flow algorithms, dynamic tree implementations, a parametric maximum flow algorithm, and algorithms for the minimum-cost flow problem. Column 4 of Table 1.1 summarizes the improvements obtained using this approach.

In the presentation to follow, we assume some familiarity with preflow-push algorithms and we omit many details, since they are straightforward modifications of known results. The reader interested in further details is urged to consult the appropriate paper or papers discussing the corresponding result for general networks or the book [1] or the survey paper [18].

2. Preliminaries.

2.1. Network definitions. Let $G = (V, E)$ be a directed bipartite network. We associate with each edge (v, w) in E a finite real-valued *capacity* $u(v, w)$. Let $U = \max\{u(v, w) : (v, w) \in E\}$. Let source s and sink t be the two distinguished vertices in the network. We make the assumption that $s \in V_2$ and $t \in V_1$. We further assume, without loss of generality, that if (v, w) is in E then so is (w, v), and that the network contains no parallel edges. We define the *edge incidence list* $I(v)$ of a vertex $v \in V$ to be the set of edges directed out of vertex v, i.e., $I(v) = \{(v, w) : (v, w) \in E\}$.

2.2. Flow. A *flow* is a function $f : E \to \mathbf{R}$ satisfying

$$f(v, w) \leq u(v, w) \quad \forall (v, w) \in E, \tag{2.1}$$

TABLE 1.1

A summary of the results discussed in this paper. Column 2 contains previously known results for general graphs. Column 3 gives bounds on bipartite networks based on the improved bound on L. Column 4 gives our new results based on the two-edge push rule.

Algorithm	Running time, general network	Running time, bipartite network	Running time, modified version
Maximum Flows			
Dinic [10]	n^2m	n_1^2m	does not apply
Karzanov [25]	n^3	n_1^2n[21]	$n_1m + n_1^3$
MPM [27]	n^3	n_1^2n[21]	does not apply
FIFO preflow-push [15], [16]	n^3	n_1^2n	$n_1m + n_1^3$
Highest label preflow-push [7]	$n^2\sqrt{m}$	$n_1n\sqrt{m}$	n_1m $+ \min\{n_1^3, n_1^2\sqrt{m}\}$
Excess scaling [2]	$nm + n^2 \log U$	$n_1m + n_1n \log U$	$n_1m + n_1^2 \log U$
Wave scaling [3]	$nm + n^2\sqrt{\log U}$	$n_1m + n_1n\sqrt{\log U}$	$n_1m + n_1^2\sqrt{\log U}$
FIFO w/ dynamic trees [15], [16]	$nm \log(\frac{n^2}{m})$	$n_1m \log(\frac{n^2}{m})$	$n_1m \log(\frac{n_1^2}{m} + 2)$
Parallel excess scaling [2]	$n^2 \log U \log(\frac{m}{n})$, $\lceil m/n \rceil$ processors	$n_1n \log U \log(\frac{m}{n})$, $\lceil m/n \rceil$ processors	$n_1^2 \log U \log(\frac{m}{n_1})$, $\lceil m/n_1 \rceil$ processors
Parametric Flows			
GGT [14]	n^3	n_1n^2	n_1^2n
GGT w/ dynamic trees [14]	$nm \log(\frac{n^2}{m})$	$n_1m \log(\frac{n^2}{m})$	$n_1m \log(\frac{n_1^2}{m} + 2)$
Min-Cost Flows			
Cost scaling [17]	$n^3 \log(nC)$	$n_1^2n \log(n_1C)$	$n_1m + n_1^3 \log(n_1C)$
Cost scaling w/ dynamic trees [17]	$nm \log(\frac{n^2}{m})$ $\cdot \log(nC)$	$n_1m \log(\frac{n^2}{m})$ $\cdot \log(n_1C)$	$n_1m \log(\frac{n_1^2}{m} + 2)$ $\cdot \log(n_1C)$

$$f(v, w) = -f(w, v) \quad \forall (v, w) \in E, \tag{2.2}$$

$$\sum_{v \in V} f(v, w) = 0 \quad \forall w \in V - \{s, t\}. \tag{2.3}$$

The *value* of a flow is the net flow into the sink, i.e.,

$$|f| = \sum_{v \in V} f(v, t).$$

The *maximum flow problem* is to determine a flow f for which $|f|$ is maximum.

2.3. Preflow. A *preflow* is a function $f : E \to \mathbf{R}$ that satisfies conditions (2.1), (2.2), and the following relaxation of condition (2.3):

$$\sum_{v \in V} f(v, w) \geq 0 \quad \forall w \in V - \{s\}. \tag{2.4}$$

The maximum flow algorithms described in this paper maintain a preflow during the computation. For a given preflow f, we define, for each vertex $w \in V$, the *excess* $e(w) = \sum_{v \in V} f(v, w)$. A vertex other than t with strictly positive excess is called *active*.

2.4. Residual capacity. With respect to a preflow f, we define the *residual capacity* $u_f(v, w)$ of an edge (v, w) to be $u_f(v, w) = u(v, w) - f(v, w)$. The *residual network* is the network consisting only of edges that have positive residual capacity.

2.5. Distance labels. A distance function $d : V \to \mathbf{Z}^+ \cup \{\infty\}$ with respect to the residual capacities $u_f(v, w)$ is a function mapping the vertices to the nonnegative integers. We say that a distance function is *valid* if $d(s) = 2n_1$, $d(t) = 0$, and $d(v) \leq d(w) + 1$ for every edge (v, w) in the residual network. We call a residual edge with $d(v) = d(w) + 1$ *eligible*. The eligible edges are exactly the edges on which we push flow.

We refer to $d(v)$ as the *distance label* of vertex v. It can be shown that if the distance labels are valid, then each $d(v)$ is a lower bound on the length of the shortest path from v to t in the residual network. If there is no directed path from v to t, however, then $d(v)$ is a lower bound on $2n_1$ plus the length of the shortest path from v to s. If, for each vertex v, the distance label $d(v)$ equals the minimum of the length of the shortest path from v to t and $2n_1$ plus the length of the shortest path from v to s, then we call the distance labels *exact*.

3. The generic preflow-push algorithm on bipartite networks. All maximum flow algorithms described in this paper are *preflow-push algorithms*, i.e., algorithms that maintain a preflow at every stage. They work by examining active vertices and pushing excess from these vertices to vertices estimated to be closer to t. If t is not reachable, however, an attempt is made to push the excess back to s. Eventually, there will be no excess on any vertex other than t. At this point the preflow is a flow, and moreover it is a maximum flow [15], [16]. The algorithms use distance labels to measure the closeness of a vertex to the sink or the source.

The generic preflow-push algorithm consists of a preprocessing stage followed by repeated application of a procedure called *push/relabel*. These two procedures appear in Fig. 3.1.

```
procedure preprocess
begin
    f = 0;
    push $u(s, v)$ units of flow on each edge $(s, v) \in I(s)$;
    compute the exact distance label function $d$ by
        backward breadth-first searches from $t$ and from $s$
        in the residual network;
end
procedure push/relabel($v$)
begin
    if there is an eligible edge $(v, w)$
    then
        begin select an eligible edge $(v, w)$;
            push $\delta = \min\{e(v), u_f(v, w)\}$ units of flow from $v$ to $w$
        end
    else replace $d(v)$ by $\min\{d(w) + 1 : (v, w) \in I(v)$ and $u_f(v, w) > 0\}$
end
```

FIG. 3.1. *Two procedures for the generic preflow-push algorithm.*

Increasing the flow on an edge is called a *push* through the edge. We say a push of δ units of flow on edge (v, w) is *saturating* if $\delta = u_f(v, w)$ and *nonsaturating* otherwise. A nonsaturating push at vertex v reduces $e(v)$ to zero. We refer to the process of increasing the distance label of a vertex as a *relabel* operation. The purpose of the relabel operation is to create at least one eligible edge on which the algorithm can perform further pushes.

Not specified in Fig. 3.1 is an efficient way to choose edges for pushing steps. We assume the same mechanism as that proposed by Goldberg and Tarjan [15], [16]. The algorithm maintains the incidence list $I(v)$ for each vertex v, and a pointer into each such list indicating a *current edge*. Initially the current edge of each incidence list is the first edge on the list. To perform *push/relabel*(v), the current edge pointer for v is moved through the list $I(v)$ until it indicates an eligible edge or it reaches the end of the list. In the former case, a push is done on the current edge. In the latter case, a relabel of v is done and the pointer is reset to indicate the first edge on $I(v)$. Figure 3.2 contains the algorithm *preflow-push*, which combines the two subroutines of Fig. 3.1. At the termination of the algorithm, each vertex in $V - \{s, t\}$ has zero excess; thus the final preflow is a flow. It is easy to establish that this flow is maximum. We shall briefly discuss the worst-case time complexity of the algorithm. (We refer the reader to the paper of Goldberg and Tarjan [16] for a complete discussion of the algorithm.)

```
algorithm preflow-push
begin
        preprocess;
        while the network contains an active vertex do
        begin
            select an active vertex $v$;
            push/relabel($v$)
        end
end
```

FIG. 3.2. *Algorithm* preflow-push.

We begin by stating two lemmas from [15] and [16].

LEMMA 3.1 [15], [16]. *The generic preflow-push algorithm maintains valid distance labels at each step. Moreover, each relabeling of a vertex v strictly increases $d(v)$.*

LEMMA 3.2 [15], [16]. *At any time during the preflow-push algorithm, for each vertex v with positive excess, there is a directed path from vertex v to vertex s in the residual network.*

Now we can derive the necessary results specific to bipartite networks.

COROLLARY 3.3. *For each active vertex v, $d(v) \leq 4n_1$.*

Proof. When a vertex v is relabeled, it has positive excess, and hence the residual network contains a path P from v to s. Since the vertices on this path are alternately in V_1 and V_2, the maximum possible length of the path is $2n_1$. Since $d(s) = 2n_1$ and, for every edge (w, x) on P, $d(w) \leq d(x) + 1$, it must be the case that $d(v) \leq d(s) + 2n_1 = 4n_1$. □

COROLLARY 3.4. *The number of relabel steps is $O(n_1 n)$. Further, the time spent performing relabels is $O(n_1 m)$. The time spent scanning edges while finding eligible edges on which to push flow is also $O(n_1 m)$.*

Proof. The first statement follows directly from Lemma 3.1 and Corollary 3.3. The second statement follows from the fact that in order to relabel a vertex v, we must look at all of the edges in $I(v)$. Hence, we can bound the total relabeling time by $O((\sum_{v \in V} |I(v)|)(4n_1)) = O(n_1 m)$. The same bound holds for the time spent finding edges on which to push flow. □

COROLLARY 3.5. *The preflow-push algorithm performs $O(n_1 m)$ saturating pushes.*

Proof. Between two consecutive saturating pushes on an edge (v, w), both $d(v)$ and $d(w)$ must increase by 2. By Lemma 3.1 and Corollary 3.3, only $O(n_1)$ saturating pushes can be done on (v, w). Summing over all edges gives the bound. □

LEMMA 3.6. *The preflow-push algorithm performs $O(n_1^2 m)$ nonsaturating pushes.*

Proof. Omitted. (Analogous to the proof of Lemma 3.10 in [16].) □

```
procedure bipush/relabel($v$)
begin
    if there is an eligible edge $(v, w)$
    then
        begin select an eligible edge $(v, w)$;
            if there is an eligible edge $(w, x)$
            then
                begin select an eligible edge $(w, x)$;
                    push $\delta = \min\{e(v), u_f(v, w), u_f(w, x)\}$ units of flow
                        along the path $v - w - x$
                end
            else replace $d(w)$ by $\min\{d(x) + 1 : (w, x) \in I(w)$ and $u_f(w, x) > 0\}$
        end
    else replace $d(v)$ by $\min\{d(w) + 1 : (v, w) \in I(v)$ and $u_f(v, w) > 0\}$
end
```

FIG. 3.3. *The procedure* bipush/relabel.

The results in column 3 of Table 1.1 for preflow-push algorithms all follow from the known results by using Corollaries 3.4 and 3.5 to replace certain $O(n)$ bounds in the general case with $O(n_1)$ bounds in the bipartite case. Since all these results are straightforward to obtain and are dominated by those in column 4, we omit their derivations and move on to the more interesting results in column 4.

4. The bipartite preflow-push algorithm. The basic idea behind the bipartite preflow-push algorithm is to perform bipushes from vertices in V_1. A *bipush* is a push over two consecutive eligible edges; it moves excess from a vertex in V_1 to another vertex in V_1. This approach has all the advantages of the usual approach, and the additional advantage that it leads to improved running times. This approach ensures that no vertex in V_2 ever has any excess. Since all the excess resides at vertices in V_1, it suffices to account for the nonsaturating bipushes emanating from vertices in V_1. Since $|V_1| \leq |V_2|$, the number of nonsaturating bipushes is reduced.

The bipartite preflow-push algorithm is a simple generalization of the generic preflow-push algorithm. The bipartite algorithm is the same as the generic algorithm given in §3 except that the procedure *bipush/relabel* appearing in Fig. 3.3 replaces the procedure *push/relabel* in the original algorithm. The algorithm identifies eligible edges emanating from a vertex using the current edge data structure described earlier.

We call a push of δ units on the path $v - w - x$ a *bipush*. The bipush is *saturating* if $\delta = \min\{u_f(v, w), u_f(w, x)\}$ and *nonsaturating* otherwise. Observe that a nonsaturating bipush reduces the excess at vertex v to zero. The following lemma is an easy consequence of the two-edge push rule implemented in *bipush/relabel*.

LEMMA 4.1. *During the execution of the bipartite preflow-push algorithm, all excess remains on the vertices in* V_1.

Proof. The first thing the algorithm does is to saturate all edges leaving s. Since $s \in V_2$, the claim is true immediately after this step. All the other pushes in the algorithm are done using the procedure *bipush/relabel*, which pushes from a vertex in V_1 through a vertex in V_2 to another vertex in V_1, never leaving any excess on a vertex in V_2. No other operations create excess at any vertex. □

As in the original preflow-push algorithm, the bipartite preflow-push algorithm always pushes flow on eligible edges and relabels a vertex only when there are no eligible edges emanating from it. Hence Lemma 3.1 holds for this algorithm too. Lemma 3.2 also holds. Corollary 3.3 holds for vertices in V_1, but a modified version holds for vertices in V_2: if $v \in V_2$, then either $d(v) \leq 4n_1 + 1$ or $d(v) = \infty$. Corollary 3.4 holds as stated. Corollary 3.5

translates into a bound of $O(n_1 m)$ saturating bipushes. The Lemma 3.6 bound of $O(n_1^2 m)$ on nonsaturating pushes becomes a bound of $O(n_1^2 m)$ on nonsaturating bipushes. Thus we get the following result.

THEOREM 4.2. *The bipartite preflow-push algorithm runs in $O(n_1^2 m)$ time.*

We now define the concept of a *vertex examination*. In an iteration, the generic bipartite preflow-push algorithm selects an active vertex v and performs a saturating bipush or a nonsaturating bipush or relabels a vertex. In order to develop more efficient algorithms, we incorporate the rule that whenever the algorithm selects an active vertex $v \in V_1$, it keeps pushing flow from that vertex until either its excess becomes zero or it is relabeled. Consequently, there may be several saturating bipushes followed either by a nonsaturating bipush or a relabel operation; there will in general also be relabelings of vertices in V_2. We associate this sequence of operations with a vertex examination. We shall henceforth assume that the bipartite preflow-push algorithm follows this rule.

5. Specific implementations of the bipartite preflow-push algorithm. The bottleneck in the bipartite preflow-push algorithm is the time spent doing nonsaturating bipushes. There are two orthogonal approaches to reducing this time. One approach is to reduce the number of nonsaturating bipushes by selecting the vertices for *bipush/relabel* operations cleverly. We shall consider several such selection rules in §§5.1–5.4. The second approach is to reduce the time spent per nonsaturating bipush. The idea is to use a sophisticated data structure in order to push flow along a whole path in one step, rather than pushing flow along a single pair of edges. We shall study this approach in §5.5. Finally, in §5.6 we study a parallel implementation of one version of the bipartite preflow-push algorithm.

5.1. The first-in first-out (FIFO) algorithm. The FIFO preflow-push algorithm examines active vertices in first-in, first-out (FIFO) order. The algorithm maintains a queue Q of active vertices. It selects a vertex v from the front of Q and performs pushes from v while adding newly active vertices to the rear of Q. The algorithm examines v until either it becomes inactive or it is relabeled. In the latter case, v is added to the rear of Q. The algorithm terminates when Q is empty. Goldberg and Tarjan [17] showed that the FIFO algorithm performs $O(n^3)$ nonsaturating pushes. We show, using a similar analysis, that the number of nonsaturating bipushes in the bipartite case is $O(n_1^3)$.

For the purpose of the analysis, we partition the sequence of vertex examinations into several *passes*. The first pass consists of examining the vertices that become active during the *preprocess* step. For $k \geq 2$, the kth pass consists of examining all vertices that were added to the queue during the $k - 1$st pass.

LEMMA 5.1. *The number of passes over Q is $O(n_1^2)$.*

Proof. Let $\Phi = \max\{d(v) | v \text{ is active}\}$. The initial value of Φ is at most $4n_1$. Consider the effect that a pass over Q can have on Φ. If, during the pass, no vertex in V_1 is relabeled, then the excess at every vertex is pushed to a vertex with a distance label smaller by at least two, and consequently Φ decreases by at least two. If some vertex in V_1 is relabeled during the pass, however, then Φ can increase or remain the same. In such a case the increase in Φ is bounded by the largest increase in any distance label. Hence, by Corollary 3.3, the total increase in Φ over all passes is at most $4n_1^2$. Consequently, the total number of passes is $O(n_1^2)$. □

Now observe that any pass examines each vertex in V_1 at most once and each vertex examination performs at most one nonsaturating bipush. Consequently, the algorithm performs $O(n_1^3)$ nonsaturating bipushes. We noted in the previous section that all other operations take $O(n_1 m)$ time. Thus we obtain the following result.

THEOREM 5.2. *The bipartite* FIFO *preflow-push algorithm runs in $O(n_1m + n_1^3)$ time.*

We note that this bound is also achieved by Karzanov's algorithm [25] if it is implemented using the two-edge push rule. A modification of Karzanov's algorithm by Tarjan [36], which he calls the *wave algorithm*, also has the same time bound. The analysis of both of these algorithms is straightforward and hence omitted.

5.2. The highest-label preflow-push algorithm. The highest-label preflow-push algorithm always pushes from an active vertex with highest distance label. This rule can be implemented using a simple bucketing approach so that the overhead for vertex selection is $O(n_1^2)$. The nonsaturating bipushes performed by the algorithm can be divided into passes. A *pass* consists of all bipushes that occur between two consecutive relabel steps of vertices in V_1. Within a pass, vertices in V_2 can possibly be relabeled several times. Notice that in this algorithm, excesses that are most distant from the sink are pushed down two levels at a time. Consequently, if the algorithm does not relabel any vertex during n_1 consecutive vertex examinations, all excess reaches the sink and the algorithm terminates. Since the algorithm performs $O(n_1^2)$ relabel operations on vertices in V_1, we immediately obtain a bound of $O(n_1^3)$ on the number of vertex examinations. As each vertex examination entails at most one nonsaturating bipush, this gives a bound of $O(n_1^3)$ on the number of nonsaturating bipushes and a bound of $O(n_1m + n_1^3)$ on the running time of the algorithm.

Cheriyan and Maheshwari [7] showed by a clever argument that the highest label preflow-push algorithm performs $O(n^2\sqrt{m})$ nonsaturating pushes for general networks. Modifying their argument to fit the bipartite case, we obtain a running time of $O(n_1m + \min\{n_1^3, n_1^2\sqrt{m}\})$. This improves the above bound of $O(n_1m + n_1^3)$ if $\sqrt{m} < n_1$. We shall give a potential-based argument that is slightly different from the analysis of Cheriyan and Maheshwari.

We focus on the set of edges that are both current and eligible; we call these edges *live*. Recall that an edge (v, w) is eligible if it has positive residual capacity and $d(v) = d(w) + 1$; (v, w) is current if the current edge pointer for vertex v indicates (v, w). Each vertex has at most one outgoing live edge, and the live edges form no cycles since $d(v) > d(w)$ if (v, w) is a live edge. Thus the set of live edges defines a forest, which we call the *live forest*. We call an active vertex *maximal* if it has no active proper descendant in the live forest. For a vertex v, let $desc(v)$ be the number of descendants of v in the live forest, including v itself, that are in V_1. Let p be a positive integer parameter whose value we shall choose later. For a maximal active vertex v, we define the *uncounted cost* $c(v)$ of v to be $\min\{0, desc(v) - p\}$. For any vertex v that is not maximal active, we define $c(v) = 0$. We use the sum $\sum_{v \in V_1} c(v)$ to help bound the number of nonsaturating bipushes.

We wish to count nonsaturating bipushes. Our strategy is to charge nonsaturating bipushes against changes in current edges, relabelings, increases in the total uncounted cost, and certain other events. We shall obtain an overall bound of $O(n_1mp + n_1^3/p)$ on the number of nonsaturating bipushes. Choosing $p = \max\{1, \lceil n_1/\sqrt{m}\rceil\}$ then gives a bound of $O(\min\{n_1m + n_1^3, n_1^2\sqrt{m}\})$ on the number of nonsaturating bipushes.

Define a *pass* of the algorithm to be a maximal interval of time during which all vertices selected for *bipush/relabel* steps have the same distance label. A pass terminates either when a relabeling occurs or when all excess at vertices with maximum distance label is moved to vertices of distance label lower by two.

LEMMA 5.3. *The total number of nonsaturating bipushes is $O(n_1mp + n_1^3/p)$.*

Proof. An argument like that in Lemma 5.1 shows that the total number of passes is $O(n_1^2)$. Consider the nonsaturating bipushes that occur during a pass. Every vertex from which a bipush occurs is maximal active. For a vertex v, call a nonsaturating bipush from v *large* if $c(v) = 0$ before the bipush and *small* otherwise. Two vertices v and w from which

```
algorithm bipartite excess scaling
begin
    preprocess;
    $\Delta = 2^{\lceil \log U \rceil}$;
    while $\Delta \geq 1$ do
    begin
        while the network contains a vertex $v \in V_1$
            with excess greater than $\Delta/2$ do
        begin
            among vertices with excess exceeding $\Delta/2$,
                select a vertex $v$ with smallest distance label;
            perform bipush/relabel($v$)
                (modified to ensure that no excess exceeds $\Delta$)
        end;
        $\Delta = \Delta/2$
    end
end
```

FIG. 5.1. Bipartite excess scaling *algorithm.*

nonsaturating bipushes occur during the pass have disjoint sets of descendants in the live forest. If a large bipush occurs from a vertex v, v has at least p V_1-descendants before the bipush. Since the total number of vertices in V_1 is n_1, there can be at most n_1/p large bipushes during the pass.

The following argument shows that every small nonsaturating bipush causes an increase of at least one in the total uncounted cost. Consider such a bipush from a vertex v to a vertex x. The bipush causes vertex v to become inactive and may cause vertex x to become maximal active; no other vertex can become maximal active. If x becomes maximal active, the total uncounted cost increases by at least one, because $desc(x) > desc(v)$ and $desc(v) < p$. If x does not become maximal active, then the total uncounted cost still increases by at least one, since the negative term $desc(v) - p$ is removed from the total uncounted cost.

We conclude that there are $O(n_1^3/p)$ nonsaturating bipushes (the large ones) plus those accounted for by increases in the total uncounted cost. It remains to bound the sum of all increases in the total uncounted cost. The total uncounted cost remains between $-pn_1$ and zero. A nonsaturating bipush cannot decrease the total uncounted cost. A saturating bipush or a relabeling or a change in a current edge can reduce the total uncounted cost by at most $O(p)$, since any such operation affects only $O(1)$ maximal active vertices. We conclude that the sum of all decreases in the total uncounted cost is $O(n_1 mp)$, and so is the sum of all increases in the total uncounted cost. The lemma follows. □

THEOREM 5.4. *The highest label preflow-push algorithm runs in* $O(n_1 m + \min\{n_1^3, n_1^2\sqrt{m}\})$ *time.*

Proof. Immediate from Lemma 5.3 by choosing $p = \max\{1, \lceil n_1/\sqrt{m}\rceil\}$. □

5.3. The excess scaling algorithm. The *excess scaling* algorithm, due to Ahuja and Orlin [2], incorporates scaling of the excesses into the generic preflow-push algorithm, thereby reducing the number of nonsaturating pushes from $O(n^2 m)$ to $O(n^2 \log U)$. The basic idea is to push flow from active vertices with sufficiently large excess to vertices with sufficiently small excess while never letting the excesses become too large. We shall develop an adaptation of the excess scaling algorithm for bipartite networks, which we call the *bipartite excess scaling algorithm.* This algorithm, in contrast to the algorithms in §§5.1 and 5.2, requires that the edge capacities be integral.

Fig. 5.1 describes the bipartite excess scaling algorithm. The algorithm uses the same *bipush/relabel* step as the generic bipartite preflow-push algorithm but with one slight difference. If $x \neq t$, instead of pushing $\delta = \min\{e(v), u_f(v, w), u_f(w, x)\}$ units of flow, it pushes $\delta = \min\{e(v), u_f(v, w), u_f(w, x), \Delta - e(x)\}$ units, where Δ is a positive *excess bound* maintained by the algorithm. This change ensures that the algorithm permits no excess on an active vertex to exceed Δ units. Since Δ is integral until the algorithm terminates, all excesses remain integral, which implies that on termination only s and t can have nonzero excess. This implies that the algorithm is correct.

LEMMA 5.5. *The bipartite excess scaling algorithm maintains the following three invariants:*

1. *No vertex in V_2 ever has positive excess.*
2. *Any bipush that does not saturate an edge moves at least $\Delta/2$ units of flow.*
3. *No vertex ever has excess greater than Δ.*

Proof. Invariant 1 is satisfied because the bipartite excess scaling algorithm is a special case of the generic algorithm and the generic algorithm satisfies it. For invariants 2 and 3, see [2] and [3]. □

We can use these invariants to establish a bound on the number of nonsaturating bipushes. We define a *scaling phase* to be a maximal period of time during which Δ does not change.

LEMMA 5.6. *The bipartite excess scaling algorithm performs $O(n_1^2 \log U)$ nonsaturating pushes and runs $O(n_1 m + n_1^2 \log U)$ time.*

Proof. As in [3], we consider the potential function $\Phi = \sum_{v \in V} \frac{e(v)d(v)}{\Delta}$, which by invariant 1 is the same as $\sum_{v \in V_1} \frac{e(v)d(v)}{\Delta}$. By invariant 3, at the beginning of a scaling phase, $\Phi \leq 4n_1^2$. The actions of the algorithm consist of bipushes and relabels. We consider the two cases separately.

Case 1. A relabel occurs. If a vertex in V_2 was relabeled, Φ remains unchanged. If a vertex in V_1 was relabeled, Φ increases by at least one. By Corollary 3.3, such increases sum to $O(n_1^2)$. (This bound actually applies to the whole algorithm, not just one scaling phase.)

Case 2. A bipush occurs. This must decrease Φ. If the bipush is nonsaturating, then by invariant 2, it moves at least $\frac{\Delta}{2}$ units of flow to a vertex with distance label two units lower, so Φ decreases by at least 1. As the initial value plus the total increase to Φ are $O(n_1^2)$, Φ can decrease by $O(n_1^2)$ per scaling phase, which means there are $O(n_1^2)$ nonsaturating pushes per scaling phase.

Observe that originally $\Delta < 2U$, where U is the maximum capacity in the network, and that when Δ decreases below 1, the algorithm terminates. In each scaling phase, Δ decreases by a factor of 2, so there are $O(\log U)$ scaling phases. Thus the total number of nonsaturating pushes is $O(n_1^2 \log U)$.

The running time of the algorithm is $O(n_1 m + n_1^2 \log U)$ plus the time required to select the smallest distance vertices for *push/relabel* steps. The bucket-based data structure described in [3] makes the total time for vertex selection $O(n_1 m + n_1^2 \log U)$. □

5.4. Variants of excess scaling. Ahuja, Orlin, and Tarjan [3] have developed two variants of the excess scaling algorithm that achieve improved time bounds. The faster of these, called the *wave scaling algorithm*, runs in $O(nm + n^2\sqrt{\log U})$ time. The idea of bipushes can easily be incorporated into both of their algorithms, thereby improving the running times for bipartite networks. The following theorem states the running time of the bipartite wave scaling algorithm.

THEOREM 5.7. *The bipartite wave scaling algorithm runs in $O(n_1 m + n_1^2\sqrt{\log U})$ time.*

The derivation of this time bound is similar to that of the excess scaling algorithm. The analysis of the original algorithm uses arguments based on potential functions defined over the vertex set V. For bipartite networks, we define the potential functions over the set V_1 and are able to replace n by n_1 in the running time. The detailed proof of this theorem is quite lengthy but contains no new ideas; therefore we omit it. A similar improvement can be obtained in Ahuja, Orlin, and Tarjan's less efficient algorithm, called the *stack scaling algorithm*.

5.5. Dynamic trees. In the previous four sections, we reduced the time needed to compute a maximum flow by reducing the number of nonsaturating pushes. In this section, we consider a different approach: we reduce the time spent per nonsaturating push. The idea is to use a sophisticated data structure in order to push flow along a whole path in one step, rather than pushing flow along a single edge. The *dynamic tree* data structure of Sleator and Tarjan [34], [33], [37] is ideally suited for this purpose.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each edge of which has an associated real value. We adopt the convention that tree edges are directed towards the root. We denote the parent of v by $p(v)$ and regard each vertex as an ancestor and descendent of itself. We call a dynamic tree *trivial* if it contains only one V_2-vertex and *nontrivial* otherwise. The data structure supports the operations in Fig. 5.2. It is shown in [34] that if the maximum number of vertices in any tree is k, we can perform an arbitrary sequence of l tree operations in $O(l \log k)$ time.

make-tree(v)	Make vertex v into a one-vertex dynamic tree.
find-root(v)	Return the root of v's tree.
find-size(v)	Return the number of vertices in v's tree.
find-value(v)	Return the value of the tree edge from v to its parent. Return ∞ if v is a root.
find-min(v)	Return the ancestor w of v with minimum *find-value*(w). In case of a tie, choose the w closest to the root. Choose v if v is the root.
change-value(v, z)	Add z to the value of every edge from v to *find-root*(v).
link(v, w, x)	Combine the trees containing v and w by making w the parent of v and giving edge (v, w) the value x. Do nothing if v and w are in the same tree or if v is not a root.
cut(v)	Break v's tree into two trees, by deleting the edge joining v and v's parent. Do nothing if v is a root.

FIG. 5.2. *Dynamic tree operations.*

In maximum flow algorithms, the dynamic tree edges are a subset of the *current edges*. The value of a tree edge is its residual capacity. We maintain the invariant that every active vertex is a dynamic tree root. For this section, we relax the invariant that all excess is on vertices in V_1 and allow excess to accumulate on vertices in V_2.

The key to the dynamic tree implementation is the *tree-push/relabel* operation in Fig. 5.3. The operation is applied to an active vertex v. If there is an eligible edge (v, w) then the operation adds (v, w) to the forest of dynamic trees, pushes as much flow as possible from v to the root of the tree containing w, and then deletes from the forest all edges which are saturated by this push. Otherwise, v is relabeled and its children are cut off. We refer to the operation of pushing flow from a node of a dynamic tree to the root as a *tree-push*.

The first dynamic tree algorithm we consider is just the generic *preflow-push* algorithm with the *push/relabel* operation replaced by the *tree-push/relabel* operation of Fig. 5.3. We modify the initialization so that each vertex is in its own one-vertex dynamic tree and we add a post-processing step which extracts the correct flow on each edge that remains in a dynamic tree. We call this algorithm the *generic bipartite dynamic tree* algorithm.

The correctness of this algorithm is straightforward to verify (see [15] and [16]). We show that this implementation yields an efficient algorithm.

```
      procedure tree-push/relabel($v$)
      begin
          if there is an eligible edge $(v, w)$
          then
              begin $link(v, w, u_f(v, w))$
                  $p(v) \leftarrow w$
                  $\delta \leftarrow \min\{e(v), find\text{-}value(find\text{-}min(v))\}$
                  $change\text{-}value(v, -\delta)$
(*)               while $v \neq find\text{-}root(v)$ and $find\text{-}value(find\text{-}min(v)) = 0$ do
                      begin $z \leftarrow find\text{-}min(v)$
(**)                      $cut(z)$
                      end
              end
          else begin replace $d(v)$ by $\min\{d(w) + 1 : (v, w) \in I(v)$ and $u_f(v, w) > 0\}$
(†)               for all children $y$ of $v$ do
(‡)                   $cut(y)$
              end
      end
```

FIG. 5.3. *The tree-push/relabel operation.*

LEMMA 5.8. *The number of* tree-push/relabel *operations done by the generic bipartite dynamic tree algorithm is $O(n_1 m)$.*

Proof. Each *tree-push/relabel* operation either relabels a vertex or pushes flow along a tree path. If it pushes flow then it must either saturate an edge or decrease the number of tree roots by one. By Corollaries 3.4 and 3.5 a relabeling or an edge saturation can occur at most $O(n_1 m)$ times. Furthermore the total increase in the number of tree roots caused by such operations is $O(n_1 m)$. Thus a push which decreases the number of tree roots by one can occur at most $O(n_1 m + n)$ times, which is the sum of the number of times the number of tree roots can increase by one plus the number of initial tree roots. □

Recalling the assumption about vertex examinations that bounds the time spent deciding which vertex and edge to process, we get the following theorem.

THEOREM 5.9. *The generic bipartite dynamic tree algorithm runs in $O(n_1 m \log n)$ time.*

Proof. Each call to *tree push/relabel* does $O(1)$ dynamic tree operations and then executes the *while* loop in line (*) or the *for* loop in line (†) a number of times. Each execution of the *while* loop takes $O(1)$ dynamic tree operations, and the *while* loop is executed at most $O(n_1 m)$ times over the course of the whole algorithm, since each *cut* in line (**) corresponds to a saturating push. Similarly the *cuts* in line (‡) correspond to edges looked at while relabeling and by Corollary 3.4 there are only $O(n_1 m)$ of these. Thus the algorithm performs $O(n_1 m)$ dynamic tree operations. Since the maximum tree size is n, the algorithm takes $O(n_1 m \log n)$ time. □

Note that we have used the fact that the number of links, the number of cuts, the number of saturating pushes, and the relabeling time are all $O(n_1 m)$.

5.5.1. Further improvements. While for many values of n, n_1, m, and U, the bound given by Theorem 5.9 is an improvement over those of the algorithms in the previous four sections, it is possible to use dynamic trees in a more sophisticated manner to achieve a running time of $O(n_1 m \log((n_1^2/m) + 2))$. In order to realize this bound, we must overcome a few obstacles. First, as in [3], [15], and [16], we need to limit the tree size. Moreover, we need to make the tree size bound solely a function of n_1 rather than n. Finally, we must deal with the fact that a *cut* can make a V_2-vertex a tree root. This leaves open the possibility that a V_2-vertex will become active, thus violating one of the invariants we have previously maintained. We see no way to avoid this—instead we control how this happens and use a fairly complicated analysis to show that we can achieve the desired time bounds.

To ensure that the tree size is a function of n_1 and not n, we use the following.

LEMMA 5.10. *If all the leaves in a nontrivial dynamic tree are V_1-vertices, then the number of vertices in the tree is at most twice the total number of V_1-vertices in the tree.*

Proof. Since no V_2-vertex is a leaf, all V_2-vertices have at least one child. The graph is bipartite, which means that all these children must be V_1-vertices. Therefore, the total number of V_1 vertices in the tree must be at least as large as the total number of V_2-vertices. □

We will use two rules to enforce this invariant. First, if a *link* operation could make a V_2-vertex a leaf, we do not perform that *link*. This rule will be respected in all the procedures that follow. Second, if a *cut* causes a V_2-vertex to become a leaf, we immediately cut that vertex from the tree. This idea is implemented in procedure *bi-cut*, which appears in Fig. 5.4. Procedure *bi-cut* will be used in place of *cut*. Observe that procedure *bi-cut* performs at most two dynamic tree operations.

```
procedure bi-cut($v$)
begin
    if $v \in V_1$
    then cut($p(v)$)
    cut($v$)
end
```

FIG. 5.4. *The bi-cut operation.*

We also want to maintain the invariant that no tree have more than k vertices (k will be chosen later). As in [15] and [16] we achieve this by preceding each *link* operation by a calculation of whether or not the result of the link will be a tree of greater than k vertices. If so, we do not perform the link. Since trees only grow as the result of *link* operations, it is clear that this maintains the desired invariant.

The main problem left to address is the complexity added by allowing excess to remain on V_2-vertices. In general, this yields slower running times. We maintain the following invariant, however.

INVARIANT 5.11. *Whenever a V_2-vertex is relabeled, it does not have any excess on it.*

As we shall see, this will allow us to get a good bound on the number of tree operations.

To maintain this invariant we need to ensure that we always have the flexibility to send all the excess from a V_2-vertex out over the current edge. The following lemma gives a condition sufficient to guarantee this flexibility.

LEMMA 5.12. *Let out-cap(v) be the residual capacity of the current edge of v. If for all V_2-vertices v that are dynamic tree roots, we maintain that*

$$e(v) \leq \textit{out-cap}(v) \tag{5.1}$$

and that the current edge of v is eligible, then Invariant 5.11 *can be satisfied with* $O(1)$ *additional work per tree-push or relabeling operation.*

Proof. The left side of (5.1) can change when we do a push that involves v, and the right side can change when the current edge of v changes. We deal with these two cases separately. When doing a tree-push that terminates at a root r that is a V_2-vertex we must ensure that the new excess does not exceed *out-cap*(r). To do this we simply push less flow. This idea is captured in a new procedure called *bi-send*, which appears in Fig. 5.5. This procedure will be used whenever we want to push flow along a path from a tree vertex to the root.

```
procedure bi-send(v)
begin
        $f \leftarrow$ find-root(v)
        if $r \in V_1$
        then
                $\delta \leftarrow \min\{e(v), \textit{find-value}(\textit{find-min}(v))\}$
        else
(*)             $\delta \leftarrow \min\{e(v), \textit{find-value}(\textit{find-min}(v)), \textit{out-cap}(r) - e(r)\}$
        change-value(v, $-\delta$)
        while $v \neq$ find-root(v) and find-value(find-min(v)) $= 0$ do
                begin $z \leftarrow$ find-min(v)
                        bi-cut(z)
                end
end
```

FIG. 5.5. *The bi-send operation.*

Next we have to deal with the case when *out-cap*(v) changes. Let (v, w) be the current edge of v. The value of *out-cap*(v) may change in two different ways. One way is that (v, w) may become saturated. When this happens, invariant (5.1) implies that the push saturating (v, w) rids v of all its excess. After the push, we advance the current edge pointer of v to the next eligible edge, doing a relabeling if necessary. The second case is that w may be relabeled, thus making (v, w) ineligible. The current edge pointer of v is advanced to the next eligible edge; for this new edge, (5.1) may be violated, however. To handle this case, we always push flow over edge (v, w) before relabeling w. This change is summarized in procedure *bi-relabel*(w), which appears in Fig. 5.6. Observe that since all edges incident to w must be inspected in order to relabel w, procedure *bi-relabel* runs in the same asymptotic time as procedure *relabel*.

```
procedure bi-relabel(w)
begin
        if $w \in V_1$
        then for all $v$ s.t. the current edge of $v$ is $(v, w)$ do
                        push $e(v)$ units of flow over edge $(v, w)$
        replace $d(v)$ by $\min\{d(w) + 1 : (v, w) \in I(v)$ and $u_f(v, w) > 0\}$
        for all children $y$ of $v$ do
                bi-cut(y)
end
```

FIG. 5.6. *The bi-relabel operation.*

What we have shown is that whenever the current edge pointer of $w \in V_2$ advances, there is no excess at w. Since this pointer advances to the end of the list before a relabel, it must be true that at the time of a relabel there is no excess on w. Further, the only algorithmic changes are the change in line (*) of *bi-send*, which adds $O(1)$ work per tree push, the change in *bi-relabel*, which adds $O(1)$ work per relabel, and a change in the current edge advancement procedure, to make sure that current edges from V_2-vertices are always eligible. □

Given these building blocks we can give the procedure *bi-tree push/relabel*, which incorporates all of these ideas. The procedure appears in Fig. 5.7. The basic idea is similar to that used in [3], [15], and [16], in that we do a tree-push, but only perform a *link* if the size of the resulting tree is not too large. We also have the additional constraint of not performing a *link* that will cause a V_2-vertex to become a leaf. This leads to lines (T1) through (T2) of *bi-tree push/relabel* which handle the case when we are pushing from a trivial dynamic tree. In this case we first push flow over v's eligible edge (v, w). Then we do a *bi-send*(w) and proceed as if we had started at the root of w's dynamic tree. We also make one technical change and use a procedure called *bi-send** instead of *bi-send* in line (TB). Procedure *bi-send** differs from *bi-send* in that it defers doing its *cuts* until line (**) of procedure *bi-tree push/relabel.* This is done in order to avoid the case that the *link* performed in line (†) is linking a trivial dynamic tree, as this would make a V_2-vertex a leaf. (This is done purely for ease of presentation and is not necessary.)

```
procedure bi-tree-push/relabel($v$)
begin
        if there is an eligible edge ($(v, w)$)
(T1)    then begin if $v$ is a trivial $V_2$ tree
                   then begin push flow on edge $(v, w)$
                              $r \leftarrow$ find-root($w$)
(TB)                          bi-send*($w$)
                              if there is an eligible edge $(r, q)$
                              then begin $v \leftarrow r$
                                         $w \leftarrow q$
                                   end
                              else bi-relabel($r$)
(T2)                    end

                   if find-size($v$) + find-size($w$) $\leq k$
(†)                then begin link($v$, $w$, $u_f(v, w)$)
                              $p(v) \leftarrow w$
                        end
(*)                else begin push flow on edge $(v, w)$
                              bi-send($w$)
(**)                          Perform the cuts from line (TB) (there may be none)
                        end
             end
        else
             bi-relabel($v$)
end
```

FIG. 5.7. *The bi-tree-push/relabel procedure.*

We now use procedure *bi-tree-push/relabel* in a FIFO algorithm. We call this the *FIFO bipartite dynamic tree algorithm.*

Since, by Invariant 5.11, whenever a V_2-vertex is relabeled it has no excess, we can derive a bound of $O(n_1^2)$ passes over the queue, by a proof similar to that of Lemma 5.1. Define a *vertex activation* to be the event that either a vertex with zero excess receives positive excess, or a vertex with positive excess is relabeled. This corresponds to a vertex being placed on the queue. We will need to bound the number of times this occurs.

First, we give a lemma, the proof of which is simliar to that of Lemma 5.8 and Theorem 5.9, with the additional observation that the time spent in an iteration of *bi-tree-push/relabel* is within a constant factor of the amount of work done by *tree-push/relabel.*

LEMMA 5.13. *The FIFO bipartite dynamic tree algorithm runs in* $O(n_1 m \log k)$ *time plus* $O(\log k)$ *time per vertex activation.*

All that remains is to bound the number of vertex activations. First we introduce some terminology. We denote the tree containing vertex v by T_v. We call a tree *large* if the number of nodes in the tree is at least $k/2$. As a consequence of Lemma 5.10, there are only $2n_1$ vertices in all the nontrivial dynamic trees, hence there are no more than $4n_1/k$ large trees at any time. In particular we will use the fact that there are $O(n_1/k)$ large trees at the beginning of a pass over the queue.

LEMMA 5.14. *The number of vertex activations is* $O(n_1 m + n_1^3/k)$.

Proof. By Invariant 5.11, all V_2-vertices have zero excess when relabeled, thus the only vertex activations due to relabelings are from V_1-vertices. There are at most $O(n_1^2)$ of these.

There can be only $O(n_1 m)$ vertex activations for which the corresponding *bi-tree-push/relabel* executions perform a *cut* or *link* or a saturating push in line (*).

It remains to count the vertex activations for which the corresponding invocation of *bi-tree-push/relabel* does neither a cut nor a link nor a saturating push. If this occurs then it must be that *find-size*(v) +*find-size*$(w) \geq k$, i.e., either T_v or T_w is large. We consider the two cases separately.

Suppose T_v is large. Vertex v is the root of T_v. Since the push is nonsaturating, it must rid v of all its excess. If T_v has changed since the beginning of the current pass, we charge the activation to the *link* or *cut* that most recently changed T_v. This occurs at most once per *cut* and twice per *link* for a total of $O(n_1 m)$ time overall. If T_v has not changed since the beginning of the pass, we charge the activation to T_v. There are at most $O(n_1/k)$ large trees at the start of a pass, hence this case counts for $O(n_1^3/k)$ charges overall.

Suppose T_w is large. In this case the root r of T_w may be added to the queue. As before, if T_w changed during the pass we charge the activation to the *link* or *cut* which caused it, otherwise we charge it to the large tree.

We have ignored so far the possible activations in lines (T1) through (T2). It is easy to verify that these only add a constant factor to the bounds mentioned above. The reason for adding this case is to ensure that in every iteration either a *link*, *cut*, or saturation is performed, or a large tree is involved. This additional case allows us to ensure this with no asymptotic loss in the running time of the procedure.

Combining all these cases we get $O(n_1 m + n_1^3/k)$ vertex activations. □

THEOREM 5.15. *The* FIFO *bipartite dynamic tree algorithm runs in* $O(n_1 m \log((n_1^2/m) + 2))$ *time.*

Proof. Apply Lemmas 5.13 and 5.14 and choose $k = (n_1^2/m) + 2$. □

5.6. A parallel implementation. In this section, we give a parallel implementation of the bipartite excess scaling algorithm. Our model of computation is an exclusive-read exclusive-write parallel random access machine (EREW PRAM) [13]. Our algorithm runs in $O((n_1 m)/d + n_1^2 \log U) \log d)$ time using $d = \lceil \frac{m}{n_1} \rceil$ processors, thus achieving near-optimal speedup for the given number of processors. We assume familiarity with parallel prefix operations [22] and refer the reader to [2], [16], [26], and [32] for examples of the use of parallel prefix operations in network flow algorithms. Specifically, we use the fact that using d processors and $O(\log d)$ time, we can execute the following parallel prefix operation:

> *Parallel Prefix Operation*: Given $l \leq d$ numbers $f(v_1), \ldots, f(v_l)$, compute the partial sums $f(v_1), f(v_1) + f(v_2), \ldots, f(v_1) + \ldots + f(v_l)$.

Our algorithm will be the same as the excess-scaling algorithm of §5.3 with a parallel implementation of *bipush/relabel* and a few additional data structures. The same approach was taken by Ahuja and Orlin [2] in developing a parallel version of their original excess scaling algorithm.

The first step in our algorithm is to transform the input graph so that each vertex has out-degree no greater than d. This transformation yields a graph with $O(n_1)$ V_1-vertices, $O(n_2)$ V_2-vertices and $O(m)$ edges. We achieve this by repeating the following step until it is no longer applicable:

> *splitting step*: Pick a vertex v with out-degree $k > d$. Create two new vertices v' and v'' and replace edges $(v, v_{k-d+1}) \ldots (v, v_k)$ with edges (v, v'), (v', v''), and $(v'', v_{k-d+1}) \ldots (v'', v_k)$. Edges (v, v') and (v', v'') have infinite capacity, while each edge (v'', v_k) has its capacity set equal to $u(v, v_k)$.

The splitting step creates one new V_1-vertex, one new V_2-vertex, and 2 more edges. Let $\Phi = \sum_v \max\{0, \lceil(\text{out-degree}(v) - d)/(d-1)\rceil\}$. Each splitting step reduces Φ by one. Initially $\Phi = O(n_1)$ and $\Phi \geq 0$ when the algorithm terminates. Thus, we only need to perform the splitting step $O(n_1)$ times overall, adding $O(n_1)$ vertices and $O(n_1)$ edges. Similarly, we can repeat the same step to reduce the in-degree of each vertex.

Further, we can perform this step in $O(n_1 \log m)$ time on d processors. We explain how to reduce the in-degree; the out-degree can be reduced in a similar manner. First, we lexicographically sort the list of edges by their tails. This can be done on d processors in $O(n_1 \log m)$ time using Cole's sorting algorithm [8] and Brent's theorem [6]. Next, we assign one processor to each of the last d edges on the list. In $O(\log d)$ time, we can determine if all these edges have the same tail. If so, we perform the splitting step, which can be done in $O(1)$ time on d processors. We then delete these edges from the list and continue on the remainder of the list. If they do not all have the same tail, then the last vertex on the list must have degree $\leq d$. In this case we delete all edges which have the same tail as the last edge and continue on the remainder of the list. In each iteration we either delete all the edges incident to a vertex or we process d edges. Hence there are $O(n_1 + \frac{m}{d}) = O(n_1)$ iterations, each of which can be performed in $O(\log m)$ time on d processors.

For the rest of this section, we will assume, without loss of generality, that every vertex in our graph has both in-degree and out-degree $\leq d$.

We first address the problem of implementing a bipush in parallel. In the bipush operation for the maximum flow problem, it is necessary to scan the edge list for vertex v starting with the current edge for vertex v until either an eligible edge is determined or until the edge list is exhausted. In the parallel algorithm, we will scan these edges in parallel.

We begin by introducing some terminology. Let $I(v)$ denote the set of vertices w such that (v, w) is an edge, and let $\hat{I}(v)$ denote the set of vertices w such that (v, w) is an eligible edge. Let us assume that the vertices in $I(v)$ are denoted $v_1, v_2, \ldots, v_k$, where $k = |I(v)|$. Thus the jth edge emanating from vertex v is edge (v, v_j).

For each vertex $v \in V_2$, we let $\hat{r}(v) = \sum_{w \in \hat{I}(v)} r(v, w)$, and refer to $\hat{r}(v)$ as the effective residual capacity of vertex v. Note that we can always push all of the excess out of a vertex v in V_2 prior to a relabeling of v so long as the excess does not exceed the effective residual capacity.

We define the *effective residual capacity* $\hat{r}(v, w)$ of edge (v, w) as

$$\hat{r}(v,w) = \begin{cases} 0 & \text{if } (v,w) \text{ is not eligible,} \\ r(v,w) & \text{if } (v,w) \text{ is eligible and } v \in V_2, w \in V_1, \\ \min\{r(v,w), \hat{r}(w)\} & \text{if } (v,w) \text{ is eligible and } v \in V_1, w \in V_2. \end{cases}$$

In the algorithm, we will be performing pushes from one vertex in V_1 at a time, and we will subsequently push from several vertices in V_2 in parallel. By defining the effective residual capacity for edges (v, w) as we do, we will ensure that we never push more flow into any vertex $v \in V_2$ than the effective residual capacity of v. Subsequently, all of the flow can be pushed out prior to a relabel of v.

In order to achieve the speedup desired, we cannot assign one processor to each edge of $I(v)$ in a push from vertex v. Thus, we will have to more efficiently allocate processors to edges on which we wish to push flow. In order to do so, we introduce the following four procedures. In all these procedures v is a vertex from which we wish to push δ units of flow.

We use Current(v) to denote v's current edge and store the edge lists in arrays.

1. *NextCurrent*(v, δ): if pushing δ units of flow would saturate all of v's admissible edges, then output $|I(v)| + 1$. Otherwise, output the index of the edge that will be current after pushing δ units of flow from v.

2. *NewRelabel*(v, δ): output true if NextCurrent $(v, \delta) = |I(v)| + 1$ and false otherwise.

3. *NextIncrement*(v, δ): output the amount of flow that will be sent in edge NextCurrent(v, δ) when pushing flow from v.

4. *Requirement*(v, δ): output the number of edges scanned in order to send δ units of flow from v without a relabel. It is equal to $NextCurrent(v, \delta) - Current(v) + 1$.

LEMMA 5.16. *There exists a data structure that allows us to implement each of these operations in $O(\log d)$ time on one processor.*

We defer the proof until later. Assume for now that such an implementation exists.

Using these procedures, we can implement the main operation, which we call *parallel-push*(v, δ, S). This operation tries to push up to δ units of flow from vertex v using the set S of parallel processors, and so that no relabel occurs. The implementation is straightforward, and appears in Fig. 5.8.

```
procedure Parallel-push($v, \delta, S$)
begin
      $c = Current(v)$. $k = NextCurrent(v, \delta)$. $s = |S|$
(*)   For each $i$ from $c$ to $\min(k-1, c+s-1)$ do in parallel
          send $\hat{r}(v, v_i)$ units of flow in edge $(v, v_i)$, and update $\hat{r}$.
      if $s \geq k - c + 1$ and $k \leq |I(v)|$
          then send $NextIncrement(v, \delta)$ units of flow in edge $(v, v_k)$.
      $Current(v) = NextCurrent(v, \delta)$.
      end
```

FIG. 5.8. *The procedure parallel push.*

LEMMA 5.17. *Parallel-push can be implemented in $O(\log d)$ time on d processors.*

Proof. Step (*) can be implemented by a parallel prefix operation on d processors. By Lemma 5.16 all the other steps can be implemented on 1 processor in $O(\log d)$ time. □

Part of the input to parallel-push is a set of processors. We use a procedure Allocate(v, D) to implement this.

Allocate(v, D)

input: vertex v, and D, a d-dimensional vector of demands for processors from the vertices in $I(v)$. $D(j)$ is the number of processors requested by vertex v_j.

output: The vector Processors( ), where Processors(j) is the set of processors allocated to vertex v_j.

It is straightforward to implement *Allocate* with a parallel prefix operation.

Now, we are ready to put all the pieces together to get an implementation of *parallel bipush/relabel*. This simply consists of a parallel push from v, followed by a set of parallel pushes from vertices $w \in V_2$ with excess, each of which is preceded by processor allocation. The procedure concludes by relabeling the necessary vertices. The details appear in Fig. 5.9. One detail deserves explanation. We always try to push exactly $\Delta/2$ units of flow from a vertex in V_1. This is necessary to maintain the invariant that no vertex ever accumulates more than Δ units of excess.

To begin the analysis, we bound the number of iterations of this procedure.

LEMMA 5.18. *There are $O(n_1^2 \log U)$ calls to* parallel bipush/relabel *over the course of the whole algorithm.*

Proof. Each parallel bipush/relabel in the first line either moves $\Delta/2$ units of flow or results in a relabeling. By a proof similar to that of Lemma 5.6, there are at most $O(n_1^2 \log U)$ such pushes over the whole algorithm. □

LEMMA 5.19. *Each call to* parallel bipush/relabel *takes O(# of iterations of the while loop $\times \log d +$ time spent relabeling) time on d processors.*

```
procedure parallel bipush/relabel($v$)
begin
    Parallel push($v$, $\Delta/2$, $d$)
    while $e(v_j) \neq 0$ for some $v_j \in I(v)$ do
    begin
        for each j = 1 to d do in parallel
            $D(v_j) = Requirement(v_j, e(v_j))$.
        $Allocate(v, D, d)$.
        for $i = 1$ to $d$ do in parallel
        begin
(*)         $push(v_i, e(v_i), processors(i))$.
            update data structures.
        end
    end
    create a list $L$ of indices $j$ s.t. $j \in V_2$ and $NewRelabel(v_j)$ = true.
    for each $i \in L$ do $Relabel(v_i)$.
    if $NewRelabel(v) = true$ then $relabel(v)$.
end
```

FIG. 5.9. *Procedure parallel bipush/relabel.*

Proof. By Lemma 5.16 and the fact that Allocate takes $O(\log d)$ time, each step except for the parallel push in line (*) takes $O(\log d)$ time. We know from Lemma 5.17 that a push takes $O(\log d)$ time. It is easy to see that a set of pushes which use a total of d edges can also be completed in $O(\log d)$ time; thus each iteration of the while loop takes $O(\log d)$ time. The lemma follows. □

It remains to bound the number of iterations of the while loop.

LEMMA 5.20. *The while loop is executed $O((n_1 m/d) + n_1^2 \log U)$ times over the whole algorithm.*

Proof. First we observe that each vertex in $I(v)$ may have at most one nonsaturating push from it per execution of the while loop. Lemma 5.18 implies that the number of nonsaturating pushes is at most $O(n_1^2 d \log U)$ overall. Let nsp be the number of nonsaturating pushes that have occurred since the beginning of the algorithm. Consider the potential function $F = \sum_v current(v) + nsp$. Initially $F = 0$ and at termination $F =$ (# of nonsaturating pushes) $= O(n_1^2 d \log U)$. The only way for F to decrease is by a relabel. Each relabel decreases F by at most $|I(v)|$; the total decrease is $O(n_1 m)$. So, the total increase in F over the algorithm is $O((n_1^2 d \log U + n_1 m))$. A parallel push with k processors increases F by k or results in a relabeling. Each iteration in a while loop except for the last one allocates d processors; hence it increases F by d or results in a relabeling. Ignoring the last iteration of the while loop in each call to parallel bipush/relabel, we find that there are at most $O((n_1^2 d \log U + n_1 m)/d)$ iterations of the while loop. To count the last iterations, we observe that there is one last iteration per call for a total of $O(n_1^2 \log U)$. Thus, overall there are $O(\frac{n_1 m}{d} + n_1^2 \log U)$ iterations. □

LEMMA 5.21. *The total time spent relabeling is $O(((n_1 m/d) + n_1^2 \log U) \log d)$.*

Proof. We spend a total of $O(n_1 m)$ work relabeling. However, at each relabeling step we look at d edges at a time, except for the last relabel step in a call to parallel bipush/relabel. Hence the total time is $O(\frac{n_1 m}{d} + n_1^2 \log U)$ □

Now we turn to the proof of Lemma 5.16.

Proof (of Lemma 5.16). Assume for now that $k = |I(v)|$ is a power of 2 for each vertex. We create a complete binary tree whose leaves are the indices of the vertices in I(v). The key of each leaf j in the binary tree is $\hat{r}(v, v_j)$. The key of each internal vertex of the binary tree is the sum of the keys of its descendent leaves.

Whenever a vertex v is relabeled, each vertex v_j of $I(v)$ is assigned a processor, and its binary tree is updated. The assignment of processors takes $O(\log d)$ steps per relabel. Moreover, each processor updates its binary tree in $O(\log d)$ steps.

When a push from vertex v is performed, the binary tree for vertex v must be updated. If k processors are assigned then Current(v) is increased by $\leq k$, and the updating can be accomplished with k processors in $O(\log \text{d})$ time.

In order to compute $NextCurrent(v, \delta)$, we start at the root of the binary tree for v, and we select the right child or the left child depending on whether δ is less than or greater than the key of the right child. We then recur on the selected child. We also can compute $NextIncrement$ in this manner. □

Combining all the above results, we have the following theorem.

THEOREM 5.22. *Algorithm* Bipartite Excess Scaling *with* bipush/relabel *replaced by* parallel bipush/relabel *runs in* $O(((n_1 m/d) + n_1^2 \log U) \log d)$ *time on d processors on an EREW PRAM.*

Plugging in $d = \lceil \frac{m}{n_1} \rceil$, we can restate the theorem as the following corollary.

COROLLARY 5.23. *Algorithm* Bipartite Excess Scaling *with* bipush/relabel *replaced by* parallel bipush/relabel *runs in* $O(n_1^2 \log U \log \frac{m}{n_1})$ *time on* $\lceil \frac{m}{n_1} \rceil$ *processors on an EREW PRAM.*

The work done by this algorithm is within a logarithmic factor of the running time of the sequential bipartite excess scaling algorithm.

6. Parametric maximum flow. A natural generalization of the maximum flow problem is obtained by making the edge capacities functions of a single parameter λ. This problem is known as the *parametric maximum flow problem.* We consider parametric maximum flow problems in which the capacities of the edges out of the sink are nondecreasing functions of λ, the capacities of the edges into the sink are nonincreasing functions of λ, and the capacities of the remaining edges are constant. Although this type of parameterization appears to be quite specialized, Gallo, Grigoriadis, and Tarjan [14] have pointed out that this parametric problem has many applications, in computing subgraph density and network vulnerability and in solving other problems, some of which are mentioned at the end of this section.

Let $u_\lambda(v, w)$ denote the capacity of edge (v, w) as a function of λ and suppose that we wish to solve the maximum flow problem for parameter values $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_l$. Clearly, for l different values of λ, a solution can be found using l invocations of a maximum flow algorithm. This approach takes no advantage of the similarity of the successive problems to be solved, however. Gallo, Grigoriadis, and Tarjan [14] gave an algorithm for finding the maximum flow for $O(n)$ increasing values of λ in the same asymptotic time that it takes to run the Goldberg–Tarjan maximum flow algorithm *once*. If the capacities are linear functions of λ, it is easy to show that the value of the maximum flow, when viewed as a function of λ, is a piecewise linear function with no more than $n - 2$ breakpoints. In this case, they give an algorithm for finding all of the breakpoints of this function in the same asymptotic time as it takes to run the Goldberg–Tarjan maximum flow algorithm once.

In this section we give an algorithm which for l increasing values of λ finds all l maximum flows in $O(ln + ln_1^2 + n_1^3 + n_1 m)$ time. Using the dynamic tree data structure, this algorithm runs in $O(ln + n_1 m \log((ln_1 + n_1^2)/m + 2))$ time.

We begin by giving one iteration of the algorithm, i.e., determining the maximum flow for parameter value λ_i, if the maximum flow for parameter value λ_{i-1} is given. The algorithm appears in Fig. 6.1. First, we update the capacities. The capacity of an edge leaving the source may have increased. If so, we saturate the edge, by setting its flow equal to its new capacity. The capacity of an edge leaving the sink may have decreased. If it has decreased below the flow on the edge, we decrease the flow so that it is equal to the capacity. Since $t \in V_1$ by assumption, this may create excess on vertices in V_2. Therefore, we immediately push any such excess to vertices in V_1, thus re-establishing the invariant that no excess is on vertices in V_2. The second step consists of running the bipartite FIFO algorithm in the network beginning with the current f and d. This gives us a maximum flow for the parameter value λ_i.

```
Step 1 (Update preflow)
Let $i = i + 1$
$\forall(s, v) \in E$ with $d(v) < 2n_1$, let $f(s, v) = \max\{u_{\lambda_i}(s, v), f(s, v)\}$.
$\forall(v, t) \in E$, let $f(v, t) = \min\{u_{\lambda_i}(v, t), f(v, t)\}$.
$\forall v \in V_2$ while $e(v) > 0$, do push/relabel$(v)$.
Step 2 (Find maximum flow) Run the bipartite FIFO algorithm on the network with capacities $u_{\lambda_i}$, beginning with flow $f$ and distance labels $d$.
```

FIG. 6.1. *Algorithm* parametric bipartite flow.

Remark. In applications of the parametric maximum flow problem, it may happen that $s \in V_1$ or $t \in V_2$, contrary to our assumption. Such a possibility can be handled by making minor changes to the algorithm, without affecting its running time.

Now we must prove that the algorithm is correct and efficient. We do this by means of the following lemmas.

LEMMA 6.1. *At the end of each step in the algorithm, there is no excess on any vertex in V_2.*

Proof. It suffices to restrict our attention to Step 1, since Step 2 always maintains this condition. Since by assumption $s \in V_2$, increasing the flow on edges out of s can increase the excess only on vertices in V_1. Since $t \in V_2$, decreasing the flow on edges into t may create excesses on vertices in V_2. This excess is immediately removed from vertices in V_2 by the procedure *push/relabel*, however. □

LEMMA 6.2. *Throughout all iterations of the* parametric bipartite flow algorithm, *distance labels are nondecreasing.*

Proof. We first show that updating the residual capacities and the preflow between iterations maintains the validity of the distance labels. Increasing the flow on an edge (s, v) may create a new residual edge (v, s), but since $d(v) < 2n_1$, the labeling is still valid. Decreasing the flow on edges into t does not create any new residual edges, so the distance labels are still valid. We noted earlier that procedures *push/relabel* and *bipush/relabel* maintain a valid labeling. The lemma follows. □

A consequence of Lemma 6.2 is that, over all iterations of the algorithm, each vertex is relabeled $O(n_1)$ times, and the total relabeling time is $O(n_1 m)$. Furthermore, the total number of saturating pushes over the whole algorithm is $O(n_1 m)$. We bound the number of nonsaturating bipushes in the next lemma.

LEMMA 6.3. *The algorithm performs a total of $O(ln_1^2 + n_1^3)$ nonsaturating bipushes over all l iterations.*

Proof. As in the bipartite FIFO algorithm, consider the following potential function: $\Phi = \max\{d(v) | v \text{ is active}\}$. The potential function increases due to relabelings, and this increase has already been shown to be at most $4n_1^2$. The potential function may also increase in Step 1 when the preflow is updated. But this increase is at most $O(n_1)$ per iteration, and $O(n_1 l)$ over all iterations. Thus the total number of passes over the queue is $O(ln_1 + n_1^2)$ and the total number of nonsaturating bipushes is $O(ln_1^2 + n_1^3)$. □

THEOREM 6.4. *A total of l iterations of the* parametric bipartite flow algorithm *take $O(ln + n_1 m + ln_1^2 + n_1^3)$ time.*

Proof. Each execution of Step 1 takes $O(n)$ time to update the residual capacities and flows. Getting rid of the excesses at vertices in V_2 by performing *push/relabel* steps takes

$O(n)$ time per iteration plus the time to perform saturating pushes, which is $O(n_1 m)$ time overall. Hence l executions of Step 1 take $O(ln + n_1 m)$ time. The l executions of Step 2 take a total of $O(n_1 m + ln_1^2 + n_1^3)$ time, as was shown previously. The theorem follows. □

The dynamic tree data structure can be incorporated into the parametric maximum flow algorithm to improve its computational complexity. Using the ideas described in §5.5, it can be shown that the dynamic tree implementation of the parametric maximum flow problem runs in $O(ln + n_1 m \log((ln_1 + n_1^2)/m + 2))$ time.

Often applications of the parametric maximum flow problem require that the minimum cut be determined for each of the parameter values $\lambda_1, \lambda_2, \ldots, \lambda_l$. Obviously each such minimum cut can be determined by a breadth-first search of the network, requiring $O(m)$ effort per cut. Overall this time would be $O(ml)$ and for larger values of l would be a bottleneck. In order to achieve a faster time bound we maintain exact distance labels of vertices as explained in [16]. Maintaining exact distance labels requires some additional effort but no more than $O(n_1 m)$ time over all iterations. While using this method, the *minimum cut* $(X_i, \overline{X_i})$, at the end of iteration i is defined as $X_i = \{v \in V : d(v) \geq 2n_1\}$ and $\overline{X_i} = \{v \in V : d(v) < 2n_1\}$. It may also be pointed out the minimum cuts in the parametric maximum flow problem are nested, i.e., for $\lambda_1 \leq \lambda_2 \leq \lambda_3$, with corresponding cuts $(X_1, \overline{X_1})$, $(X_2, \overline{X_2})$, $(X_3, \overline{X_3})$, we have that $X_1 \subseteq X_2 \subseteq X_3$ [12]. This property allows us to store all l cuts in $O(n + l)$ space, and recreate any one cut in $O(n)$ time.

While we have only given an algorithm for the case where the λ's are given in increasing order, actually we can solve a more general problem. Let $\kappa(\lambda)$, the *min-cut capacity function*, be the capacity of the minimum cut as a function of λ. If the edge capacities are linear functions of λ, then $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n - 2$ breakpoints. We can actually compute all of these breakpoints in $O(n^2 + n_1 m \log((nn_1/m) + 2))$ time, and can do even better if we know a priori that $l = o(n)$. This result directly follows from the results of [14] and the details appear in [35].

We conclude by noting that the bipartite parametric flow problem has many applications including multiprocessor scheduling with release times and deadlines [21], [24], 0-1 integer programming problems [29], [30], maximum subgraph density [21], finding a maximum-size set of edge-disjoint spanning trees in an undirected graph [28], [29], [30], network vulnerability [9], [19], partitioning a data base between fast and slow memory [11], and the sportswriter's end-of-season problem [23], [31]. For all these problems we improve on or match the best known bounds.

7. Minimum-cost circulation. In this section we examine the *minimum-cost flow problem* on bipartite networks. We consider the recent cost-scaling minimum-cost flow algorithm of Goldberg and Tarjan [17], and describe the improvement in its running time that can be obtained when it is adapted for bipartite networks. We shall be very sketchy in our description, since all the results are analogous to the results in §5.

The minimum cost flow problem is a generalization of the maximum flow problem. In this problem, each edge (v, w) has a cost $c(v, w)$. We formulate the problem as a circulation problem, since it is equivalent to other formulations. (See [1] and [18].) We assume that the costs are *antisymmetric*, i.e., $c(v, w) = -c(w, v)$ for each edge (v, w). Let $C = \max\{c(v, w) : (v, w) \in E\}$. The minimum-cost circulation problem can be formulated as follows:

$$\text{Minimize} \sum_{(v,w)\in E} c(v, w) f(v, w)$$

subject to

$$f(v,w) \leq u(v,w) \quad \forall (v,w) \in E, \tag{7.1}$$

$$f(v,w) = -f(w,v) \quad \forall (v,w) \in E, \tag{7.2}$$

$$\sum_{w \in V} f(v,w) = 0 \quad \forall v \in V. \tag{7.3}$$

A *circulation* is a function of satisfying constraints (7.1), (7.2), and (7.3). A *pseudoflow* is a function f satisfying only constraints (7.1) and (7.2). For any pseudoflow f, we define the *excess* of vertex w to be

$$e(w) = \sum_{v:(v,w) \in E} f(v,w). \tag{7.4}$$

The excess at a vertex may be positive or negative. A vertex v is called *active* if $e(v) > 0$. The residual network is defined as for the maximum flow problem. We associate with each vertex v a real-valued *price* $p(v)$. The prices correspond to linear programming dual variables. In the analysis, the prices play a role similar to that played by the distance labels in the maximum flow algorithm. The *reduced cost* of an edge (v, w) with respect to the price function p is denoted by $c_p(v, w)$ and is defined by $c_p(v, w) = c(v, w) + p(v) - p(w)$.

7.1. The cost-scaling algorithm. The cost-scaling algorithm of Goldberg and Tarjan [17], relies on the concept of *approximate optimality*. A circulation f is said to be *ϵ-optimal* for some $\epsilon > 0$ if f together with some price function p satisfies the following condition:

$$u_f(v,w) > 0 \Rightarrow c_p(v,w) \geq -\epsilon \Longrightarrow (\epsilon\text{-optimality}). \tag{7.5}$$

We refer to this condition as the *ϵ-optimality condition.* Let l be the number of edges on the longest simple cycle in the network. It can be shown that any feasible flow is ϵ-optimal for $\epsilon \geq C$ and any ϵ-optimal feasible flow for $\epsilon < 1/l$ is an optimum flow [4]. Since in a bipartite network every other vertex on a cycle must be a vertex in V_1, any ϵ-optimal feasible flow for $\epsilon < 1/(2n_1)$ is an optimum flow.

The cost-scaling algorithm treats ϵ as a parameter and iteratively obtains ϵ-optimal flows for successively smaller values of ϵ. Initially, $\epsilon = C$; on termination, $\epsilon < 1/(2n_1)$. The algorithm performs repeated cost-scaling phases, each of which consists of applying an *improve-approximation* procedure that transforms a 2ϵ-optimal circulation into an ϵ-optimal circulation. After $1 + \lceil \log(2n_1C) \rceil$ cost scaling phases, $\epsilon < 1/(2n_1)$, and the algorithm terminates with an optimal circulation. To get the algorithm started, an initial circulation can be found by using any maximum flow algorithm, such as one of those discussed in §5. A more formal description of this algorithm appears in Fig. 7.1.

Recall that in the maximum flow algorithm, we maintained the invariant that all excess was on V_1-vertices. This will be our goal in the minimum cost circulation algorithm also. The procedure *improve-approximation* given in Fig. 7.2 first converts the 2ϵ-optimal circulation it receives as input into a 0-optimal pseudoflow (lines (*) through (**)). This may leave positive excess on V_2-vertices. So we execute the while loop at line (†), which applies *push/update* operations to these vertices until they are rid of all their excess. Now we have established the invariant that the only vertices with positive excess are V_1-vertices. We will maintain this invariant for the rest of procedure *improve-approximation.* The remainder of the procedure moves flow from vertices with positive excess to vertices with negative excess. As vertices

```
algorithm cost scaling
begin
        $p = 0$; $\epsilon = C$;
        let $f$ be any initial circulation;
        while $\epsilon \geq \frac{1}{2n_1}$ do
                begin $\epsilon = \epsilon/2$
                        improve-approximation($f$, $p$, $\epsilon$);
                end
end
```

FIG. 7.1. *Algorithm* cost scaling.

```
procedure improve-approximation($f$, $p$, $\epsilon$)
begin
(*)     if $c_p(v, w) < 0$
        then begin $f(v, w) = u(v, w)$;
                        $f(w, v) = -f(v, w)$
                end;
(**)    compute vertex imbalances;
(†)     while the network contains an active $V_2$-vertex $v$ do
                push/update($v$)
(‡)     while the network contains an active vertex $v$ do
                bipush/update($v$)
end
```

FIG. 7.2. *The procedure* improve-approximation.

in V_2 may have negative excess, this will sometimes involve a one-edge push and sometimes involve a two-edge push.

We call an edge (v, w) in the residual network *admissible* if $c_p(v, w) < 0$. We define the subnetwork of G consisting solely of admissible edges to be the *admissible network*. The basic operations in the procedure are selecting active vertices, pushing flows on admissible edges, and updating vertex prices. The details of *improve-approximation*, adapted to the bipartite case, appear in Figs. 7.2 and 7.3.

```
procedure bipush/update($v$)
begin
        if there exists an admissible edge $(v, w)$
        then if $e(w) < 0$
                then push $\min\{e(w), e(v), u_f(v, w)\}$ units of flow on $(v, w)$
                else if there exists an admissible edge $(w, x)$
                        then push $\delta = \min\{e(v), u_f(v, w), u_f(w, x)\}$ units of flow
                                along the path $v - w - x$
                        else replace $p(w)$ by $\max_{(w,x) \in E_f}\{p(x) - c(w, x) - \epsilon\}$
        else replace $p(v)$ by $\max_{(v,w) \in E_f}\{p(w) - c(v, w) - \epsilon\}$
end
procedure push/update($v$)
begin
        if there exists an admissible edge $(v, w)$
        then push $\min\{e(w), e(v), u_f(v, w)\}$ flow on $(v, w)$
        else replace $p(v)$ by $\max_{(v,w) \in E_f}\{p(w) - c(v, w) - \epsilon\}$
end
```

FIG. 7.3. *The procedures* push/update *and* bi-push/update.

To identify admissible edges emanating from a vertex, the algorithm uses the same *current edge* data structure used in the preflow-push algorithm for the maximum flow problem.

A movement of flow along a path $v-w-x$ in *bipush/update* is called a *bipush*. The bipush is *saturating* if $\delta = \min\{u_f(v, w), u_f(w, x)\}$ and *nonsaturating* otherwise. The correctness and efficiency of the algorithm rest on the following results.

LEMMA 7.1. 1. *The* improve-approximation *procedure always maintains ϵ-optimality of the pseudoflow, and at termination yields an ϵ-optimal circulation.*

2. *Each vertex price never increases, and it decreases $O(n_1)$ times during an execution of the procedure.*

3. *There are $O(n_1 m)$ saturating pushes and bipushes during an execution of the procedure.*

4. *Immediately before, during, and immediately after the while loop in line (‡) of* improve-approximation, *all excess is on V_1-vertices.*

Proof. These results follow directly from the proofs of Goldberg and Tarjan [17] adapted for bipartite networks. □

As in the preflow-push algorithm, it can easily be shown that the time spent updating prices in an execution of *improve-approximation* is $O(n_1 m)$. The bottleneck in the procedure is the number of nonsaturating pushes and bipushes. Observe that there are three different types of pushes and bipushes to bound:

1. pushes in the while loop at line (†),
2. bipushes in the while loop at line (‡),
3. pushes in the while loop at line (‡).

We bound the first type by observing the all the pushes are saturating except for at most one per V_2-vertex. Therefore, there are at most n nonsaturating pushes.

To bound the second type of bipushes, we need the following lemma from [17].

LEMMA 7.2 [17]. *The admissible network remains acyclic throughout the execution of the* improve-approximation *procedure.*

The number of nonsaturating bipushes performed by the procedure depends upon the order in which active vertices are examined. Goldberg and Tarjan [17] show that the generic version of the procedure, in which active vertices are examined in an arbitrary order, performs $O(n^2 m)$ pushes for general networks. They show that a specific implementation of the generic implementation, called the *first-active method* algorithm, performs $O(n^3)$ nonsaturating pushes, as does a related method, the *wave method.* (The wave method was developed independently by Bertsekas and Eckstein [5].) We shall show that an adaptation of the first-active method for bipartite networks performs $O(n_1^3)$ nonsaturating bipushes.

The first-active method uses the acyclicity of the admissible network. As is well known, the vertices of an acyclic network can be ordered so that for each edge (v, w), v precedes w in the ordering. Such an ordering of vertices is called a *topological ordering* and can be found in $O(m)$ time. The first-active method maintains a list L of all vertices in V_1 in topological order. The algorithm examines each vertex $v \in L$ in order. If v is active, it performs *bipush/update* operations on vertex v until either it becomes inactive or $p(v)$ is updated. In the former case, the algorithm examines the next vertex on L. In the latter case, the algorithm moves v from its current position on L to the front of L, and restarts the scan of L at the front. Moving v to the front of L preserves the invariant that L is a topological order of the vertices, because immediately after v is assigned a new prices, it has no incoming admissible edges. The algorithm terminates when L is scanned in its entirety. Note that updating the price of a vertex in V_2 does not affect the topological order of vertices in V_1. On termination, no vertex can be active.

Observe that if within n_1 consecutive vertex examinations the algorithm performs no price updates then all active vertices have discharged their excesses and the algorithm obtains a flow. This follows from the fact that when vertices are examined in the topological order, active vertices push flow to vertices after them in the topological order. As there are $O(n_1^2)$ price updates of vertices in V_1, we immediately obtain an $O(n_1^3)$ bound on the number of vertex examinations. Each vertex examination entails at most one nonsaturating bipush. Consequently, the wave algorithm performs $O(n_1^3)$ nonsaturating bipushes per execution of *improve-approximation*.

Now we bound the third type of push. A push in this case is performed over an edge (v, w) such that $e(w) < 0$. There are three cases. Either the value of the push is $u_f(v, w)$ (saturating), $e(v)$ (nonfilling), or $e(w)$ (filling). For the first case we have already bounded the number of saturating pushes. In the second case, we can bound the number of nonfilling pushes by $O(n_1^3)$ by arguments similar to those for non-saturating pushes above. For filling pushes, observe that each vertex is filled at most once per iteration of *improve-approximation*; thus there are a total of n such pushes overall.

Combining the three cases, we find that the number of nonsaturating pushes and bipushes is $O(n_1^3+n)$. As all other steps take $O(n_1 m)$ time per execution of the *improve-approximation* procedure and the procedure is called $O(\log(n_1 C))$ times, we get the following result.

THEOREM 7.3. *The wave algorithm solves the minimum cost flow problem on a bipartite network is $O((n_1 m + n_1^3)\log(n_1 C))$ time.*

8. Summary and conclusions. We have considered a number of maximum flow algorithms and algorithms for other network flow problems for bipartite networks in which one side is much smaller than the other. Our work is motivated by and improves upon the work of Gusfield, Martel, and Fernandez-Baca [21]. In that paper, the authors demonstrated the importance of bipartite maximum flow problems in which one side is much smaller than the other. In addition, they showed that existing algorithms run much faster on these "unbalanced" networks.

We have extended the results of Gusfield et al. in several ways. First of all, we showed that their analysis applies to other maximum flow algorithms. In addition, we developed the concept of the *bipush* for preflow-push algorithms and showed that bipushes lead to further improvements in several algorithms for the maximum flow problem. We further generalized the results to algorithms for the parametric maximum flow problem, as well as the minimum cost flow problem. We also showed that the results apply as well to dynamic tree implementations if the dynamic tree algorithms are modified appropriately.

Although the theory in this paper has been concerned with bipartite networks, it would be just as valid for networks in which we allow edges joining two vertices in V_1. More generally, it is valid for networks in which have a small *vertex cover*. A vertex cover of a network $G = (V, E)$ is a set S of vertices such that each edge in E is incident to at least one vertex in S. A minimum vertex cover is one with the smallest number of vertices. Although it is NP-hard to determine a minimum vertex cover of a graph, it is possible to find a vertex cover in $O(n + m)$ time whose cardinality is within a factor of 2 of the cardinality of a minimum vertex cover. (Just find any maximal matching and include each of the matched vertices).

If the size of the minimum vertex cover of a graph is n_1, then all of the time bounds presented in the previous sections apply. It is easy to show that the length of the longest path in such a network is at most $2n_1$. As for bipushes they would have to be replaced as follows. Suppose G is a network, not necessarily bipartite, in which V_1 is a vertex cover. As before we maintain the invariant that each active vertex is in V_1. Suppose that v is active, and that (v, w) is eligible. If w is in V_1 then we perform a normal push. If w is not in V_1, then each edge incident to w is in V_1 and we perform a bipush. All of the results in this paper are thus easily generalized to networks with small vertex covers, and the time bounds stated in Table 1.1 apply to such networks.

It is likely that improvements could be obtained in the running times of other algorithms for network flow problems on unbalanced bipartite networks, or on networks in which the cardinality of a minimum vertex cover is small. For example, one can obtain improved running times for dynamic programming algorithms for the shortest path problem, and one can improve the running time for all pairs shortest path algorithms. We conjecture that one can also obtain improved time bounds for the b-matching problem on networks with small vertex covers. We also conjecture that one can obtain improved results for polymatroidal network flows.

Acknowledgments. We are grateful to David Shmoys for many helpful conversations and suggestions, and for a careful reading of an earlier draft of this paper.

REFERENCES

[1] R. K. AHUJA, T. L. MAGNANTI, AND J. B. ORLIN, *Network Flows, Theory, Algorithms, and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[2] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., 37 (1989), pp. 748–759.

[3] R. K. AHUJA, J. B. ORLIN, AND R. TARJAN, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.

[4] D. BERTSEKAS, *Distributed asynchronous relaxation methods for linear network flow problems*, tech. report LIDS-P-1606, Massachusetts Institute of Technology, Laboratory for Information and Decision Sciences, Cambridge, MA, 1986.

[5] D. BERTSEKAS AND J. ECKSTEIN, *Dual coordinate step methods for linear network flow problems*, Math. Programming, 42 (1988), pp. 202–243.

[6] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. ACM, 21 (1974), pp. 201–208.

[7] J. CHERIYAN AND S. N. MAHESHWARI, *Analysis of preflow push algorithms for maximum network flow*, SIAM J. Comput., 18 (1989), pp. 1057–1086.

[8] R. COLE, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[9] W. CUNNINGHAM, *Optimal attack and reinforcement of a network*, J. ACM, 32 (1985), pp. 549–561.

[10] E. DINIC, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[11] M. J. EISNER AND D. G. SEVERANCE, *Mathematical techniques for efficient record segmentation in large shared databases*, J. ACM, 23 (1976), pp. 619–635.

[12] L. R. FORD AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

[13] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proceedings of the 10th Annual ACM Symposium on Theory of Computing, ACM Press, San Diego, CA, 1978, pp. 114–118.

[14] G. GALLO, M. D. GRIGORIADIS, AND R. E. TARJAN, *A fast parametric maximum flow algorithm and applications*, SIAM J. Comput., 18 (1989), pp. 30–55.

[15] A. V. GOLDBERG, *Efficient graph algorithms for sequential and parallel computers*, Ph.D thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[16] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, J. ACM, 35 (1988), pp. 921–940.

[17] ———, *Solving minimum-cost flow problems by successive approximation*, Math. Oper. Res., 15 (1990), pp. 430–466.

[18] A. V. GOLDBERG, R. E. TARJAN, AND E. TARDOS, *Network flow algorithms*, in Paths, Flows, and VLSI-Layout, B. Korte, L. Lovász, H. Prömel, and A. Shriver, eds., Springer-Verlag, Berlin, 1990, pp. 101–164.

[19] D. GUSFIELD, *Connectivity and edge disjoint spanning trees*, Inform. Process. Lett., 16 (1983), pp. 87–89.

[20] ———, *Computing the strength of a graph*, SIAM J. Comput., 20 (1991), pp. 639–654.

[21] D. GUSFIELD, C. MARTEL, AND D. FERNANDEZ-BACA, *Fast algorithms for bipartite network flow*, SIAM J. Comput., 16 (1987).

[22] W. HILLIS AND J. G. L. STEELE, *Data parallel algorithms*, Comm. ACM, 29 (1986), pp. 1170–1183.

[23] A. Hoffman and T. Rivlin, *When is a team "mathematically" eliminated?*, in Symposium on Mathematical Programming, Princeton University Press, Princeton, NJ, 1970, pp. 391–396.

[24] W. Horn, *Some simple scheduling algorithms*, Naval Res. Logist., 21 (1974), pp. 177–185.

[25] A. V. Karzanov, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[26] T. Leighton, C. Leiserson, B. Maggs, S. Plotkin, and J. Wein, *Advanced parallel and VLSI computation*, Research Seminar Series MIT/LCS/RSS 2, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[27] V. M. Malhotra, M. Pramodh Kumar, and, S. N. Maheshwari, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Inform. Process. Lett., 7 (1978), pp. 277–278.

[28] C. S. J. A. Nash-Williams, *Edge disjoint spanning trees of finite graphs*, J. London Math. Soc., 36 (1961), pp. 445–450.

[29] J. Picard and M. Querayne, *A network flow solution to some nonlinear* 0-1 *programming problems, with applications to graph theory*, Networks, 12 (1982), pp. 141–159.

[30] ———, *Selected applications of minimum cuts in networks*, INFOR., 20 (1982), pp. 394–422.

[31] B. Schwartz, *Possible winners in partially completed tournaments*, SIAM Rev., 8 (1966), pp. 302–388.

[32] Y. Shiloach and U. Vishkin, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[33] D. Sleator and R. Tarjan, *Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652–686.

[34] ———, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[35] C. Stein, *Efficient algorithms for bipartite network flow*, undergraduate thesis, Princeton University, Dept. of Computer Science, Princeton, NJ, 1987.

[36] R. E. Tarjan, *A simple version of Karzanov's blocking flow algorithm*, Oper. Res. Lett., 2 (1981), pp. 265–268.

[37] ———, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 934–950, October 1994

003

NEW RESULTANT INEQUALITIES AND COMPLEX POLYNOMIAL FACTORIZATION *

V.Y. PAN[†]

Abstract. The author deduces some new probabilistic estimates on the distances between the zeros of a polynomial $p(x)$ by using some properties of the discriminant of $p(x)$ and applies these estimates to improve the fastest deterministic algorithm for approximating polynomial factorization over the complex field. Namely, given a natural n, positive ϵ, such that $\log(1/\epsilon) = O(n \log n)$, and the complex coefficients of a polynomial $p(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_n \neq 0$, $\sum_i |p_i| \leq 1$, a factorization of $p(x)$ (within the error norm ϵ) is computed as a product of factors of degrees at most $n/2$, by using $O(\log^2 n)$ time and n^3 processors under the PRAM arithmetic model of parallel computing or by using $O(n^2 \log^2 n)$ arithmetic operations. The algorithm is randomized, of Las Vegas type, allowing a failure with a probability at most δ, for any positive $\delta < 1$ such that $\log(1/\delta) = O(\log n)$. Except for a narrow class of polynomials $p(x)$, these results can be also obtained for ϵ such that $\log(1/\epsilon) = O(n^2 \log n)$.

Key words. complex polynomial factorization, randomized algorithms, computational complexity, resultant

AMS subject classifications. 12D05, 62P99, 68Q25, 65Y20, 65Y05

1. Introduction.

Randomization is a powerful tool for designing effective numerical and algebraic algorithms over finite fields (see, for instance, [23]), but over infinite fields, the application of randomization is usually limited to verification of polynomial identities and to avoiding singularities (see [18] and [26]). We are going to demonstrate the power of randomization in a new area, for approximate factorization (over the complex field) of a monic univariate polynomial of degree n,

$$p(x) = \sum_{i=0}^{n} p_i x^i = p_n \prod_{j=1}^{n} (x - x_j), \qquad \|p(x)\| = \sum_i |p_i|, \tag{1}$$

that is, given a set of complex coefficients $p_0, p_1, \ldots, p_n$ and a positive ϵ, we seek approximations $x_1^*, \ldots, x_n^*$ to the zeros $x_1, \ldots, x_n$ of $p(x)$ such that

$$\|p(x) - p_n \prod_{j=1}^{n} (x - x_j^*)\| \leq \epsilon. \tag{2}$$

The requirement (2) can be motivated by the observation that in practice of numerical computation, the coefficients of $p(x)$ are most frequently available only within certain truncation errors. For the worst case input polynomial $p(x)$, where $p_n = 1$, $|x_j| \leq 1$, for all j (we may satisfy this assumption by scaling x by a power of 2, thus preserving the binary length of the coefficients), (2) implies the more classical requirement that

$$|x_j - x_j^*| < \tilde{\epsilon} \quad \text{for all} \quad j, \tag{3}$$

as long as $\tilde{\epsilon} \geq 4.4(2\epsilon)^{1/n}$; $2\epsilon < 40^{-n}$ (see Proposition 11 in the appendix). Moreover, we may represent the perturbed polynomial as $p_\epsilon(x) = p(x) + \epsilon\, q(x)$ and let $x_j(\epsilon)$ denote its zeros, so that

$$x_j(0) = x_j, \quad p_\epsilon(x_j(\epsilon)) = 0, \quad j = 1, \ldots, n.$$

*Received by the editors August 13, 1992; accepted for publication (in revised form) June 1, 1993.

[†] Lehman College, City University of New York, Bronx, New York 10468. This author was supported by National Science Foundation grant CCR 9020690 and PSC-CUNY Award 662478.

We may differentiate the latter expression, ignore the term $O(\epsilon)$, and obtain that

$$x_j'(\epsilon) = -q(x_j(\epsilon))\,/\,p'(x_j(\epsilon)),$$

provided that $p'(x_j(\epsilon)) \neq 0$. If x_j is a simple zero of $p(x)$, that is, if

$$p(x_j) = 0, \qquad p'(x_j) \neq 0,$$

then we have

$$x_j(\epsilon) = x_j + \epsilon\, x_j'(v\epsilon) = x_j - \epsilon\, q(x_j(v\epsilon))\,/\,p'(x_j(v\epsilon))$$

for some v, $0 \leq v \leq 1$.

Therefore, (2) implies (3) for simple zeros of $p(x)$ and for $\epsilon/\tilde{\epsilon}$ of the order of $|p'(x_j)|$, that is, (2) implies (3) for small ϵ and for $\tilde{\epsilon}$ roughly proportional to ϵ, unless $p(x)$ has some other zeros near x_j, so that $p'(x_j)$ is close to 0.

For polynomials $p(x)$ with multiple or clustered zeros, (2) implies (3), with $\tilde{\epsilon}$ tending to be substantially larger than ϵ as $\epsilon \to 0$.

Example 1. $p(x) = x^n$ has the only zero $x_1 = x_2 = \cdots = x_n = 0$, whose multiplicity is n. $p(x) - \epsilon = x^n - \epsilon$ has the zeros $x_j^* = \epsilon^{1/n}\omega^j$, $j = 1, \ldots, n$; $\omega = \exp(2\pi\sqrt{-1}/n)$, so that $\tilde{\epsilon} = \epsilon^{1/n}$.

Example 1 suggests that, for polynomials with multiple or clustered zeros, computing the factorization under (2) is numerically better conditioned than the zero finding problem under (3).

More important for us, the same example also suggests that after a small random perturbation of the coefficients of $p(x)$, the zeros of the resulting polynomial $p(x) + \Delta(x)$, with $\|\Delta(x)\| = \epsilon$, tend to stay apart from each other at the distance at least of the order of $(1/n)\epsilon^{1/n}$ (even if they correspond to the multiple or clustered zeros of $p(x)$), so that such a perturbation promises to eliminate a major obstacle to a rapid convergence of many known algorithms for polynomial factorization, namely, those which converge slowly on the input polynomials whose zeros form clusters.

Substantiation of this intuitive argument is not straightforward, however.

In this paper we achieved some progress towards such a substantiation, by exploiting some properties of the resultant of $p(x)$ and of its derivative (or of the discriminant of $p(x)$). Based on these properties, we estimated the *mth diameter* $d_m = d_m(x_1, \ldots, x_n)$ of the set $\{x_1, \ldots, x_n\}$,

$$d_m = \min_{|T|=m} \max_{i,j \in T} |x_i - x_j|\,, \qquad m = 2, 3, \ldots, n\,, \tag{4}$$

where the minimum is was over all the cardinality m subsets of the set $\{1, 2, \ldots, n\}$ (that is, d_m denotes the minimum diameter of a subset of m elements of the set $\{x_1, \ldots, x_n\}$, so that, in particular, $d_n = \max_{1 \leq i,j \leq n} |x_i - x_j|$). More precisely, we estimated the mth diameter of the set of the zeros of the polynomials

$$p(x) + x^h \Delta \quad \text{for} \quad h = 0, 1, \ldots, n,$$

for a random complex Δ, uniformly distributed on the circle $|\Delta| = \rho$, and for a fixed small positive ρ. We have set $m = n/2$ (assuming n even) and proved that, with a probability at least $1/(2n+1)$, the reciprocal of the mth diameter of the set of the zeros of $p(x) + x^h\Delta$ was bounded by $O(1/\rho^{4/(n-2)})$ (as $\rho \to 0$) if $h = 0$, and moreover, for a large class of the input polynomials $p(x)$, the maximum of such a reciprocal over all integers h, $0 \leq h \leq n$, was bounded by $O(1/\rho^{4/(n-2)n})$ (see (18), (53), and Proposition 4, for more specific estimates).

We also examined the resulting effect of the perturbation of $p(x)$ on the computation of the factorization of $p(x)$ by means of an effective factorization algorithm based on a combination of the techniques of [25] and [19]. The resulting hybrid algorithm recursively computes a *complete numerical factorization* (2) of $p(x)$ and supports the record running time bounds for this computation, both in terms of arithmetic and Boolean (bit) operations involved. The intermediate steps compute *incomplete numerical factorizations* of $p(x)$. If the degrees of the output factors are less than, say, 1/2 of the input degree in every recursive step, then in at most $\lfloor \log_2 n \rfloor$ recursive steps, a complete numerical factorization (2) is computed, and the worst case bounds on *both* sequential and parallel running times of the entire algorithm decrease roughly by the factors of n and $n/\lfloor \log_2 n \rfloor$, respectively. This is because in the worst case for each i, recursive step i splits its input polynomial of degree $n+1-i$ into two factors of degrees 1 and $n-i$, respectively, so that $n-1$ recursive steps are required to compute a complete recursive factorization (2).

Actually, such a balanced splitting was a cornerstone of several recent successful algorithms for polynomial factorization and/or rootfinding [4]–[7], [20]. The algorithms of [4]–[6] and [20], however, only apply to the polynomials all of whose zeros are real, whereas the parallel algorithm of [17] reaches decisive parallel acceleration in the general (complex) case, but at the expense of a great increase of its processor and sequential time bounds.

Unlike the cited papers, our approach is not fully recursive, and it works for a more limited range of the output error bounds ϵ, but it does not restrict itself to the real case, and leads to parallel acceleration without blowing up the processor bound. More specifically, by using the cited lower bounds on the distances among the zeros of the slightly perturbed polynomial, $p(x) + x^h \Delta$, we prove that, with a probability bounded away from 0, for a random choice of Δ, practically the *same algorithm* splits such a perturbed polynomial into factors of degrees less than $n/2$. Thus, the randomization alone *strictly improves* the performance of the original algorithms.

The same randomization techniques can be recursively applied to split the computed factors of $p(x)$. In each new recursive step the randomization may give us less and less advantages (see §8 on the details), and we may end up with shifting to the original deterministic factorization algorithm at some recursive step at which using randomization gives us no advantage anymore. Some techniques may counter such a deficiency (see §9), and in any case, our acceleration (due to randomization) applies to the hardest recursive steps, in which we split into factors the polynomials of higher degrees.

We will formally state our main results in the form of Theorem 1, which presents our estimates for the error and for the parallel and sequential computational complexity of a single step of splitting $p(x) + \Delta$ (for a random Δ, $|\Delta| = \rho$) into factors of degrees less than $n/2$. We note that our randomization is of Las Vegas type, where no undetected errors may occur (the algorithm may output FAILURE but with a low and controlled probability).

For simplicity, we will use the arithmetic (rather than Boolean) complexity estimates, which is adequate since our deviation from the original algorithms of [25] and [19] does not affect the required precision of the computation. We will assume the customary RAM and PRAM models of sequential and parallel arithmetic computations, respectively (see [1] and [14]), and will adopt the customary notation $O_A(t, p)$, which means that $O(t)$ parallel arithmetic steps and p processors suffice in the parallel implementation of the solution algorithm. (From now on, we use the "O" notation assuming that $n \to \infty$ and $\epsilon, \rho \to 0$.) $O_A(t, p)$ also implies $O_A(ts, p/s)$, for any integer $p/s > 0$, which gives us the sequential time bound $O_A(tp, 1)$, for $s = p$ (Brent's principle [10]).

We will assume hereafter that

$$\|p(x)\| = 1, \tag{5}$$

$$(6) \qquad 1/4 \le r(C, p(x)) = \max_j |x_j - C| \le 1 \,, C = (1/n) \sum_j x_j = -p_{n-1}/(np_n).$$

We may closely approximate $r(C, p(x))$ at the cost $O_A(\log n, n)$ (see Proposition 5), and if $r(C, p(x)) > 1$ or if $0 < r(C, p(x)) < 1/4$, we may shift, also at the cost $O_A(\log n, n)$ (see [2]), to a polynomial

$$q(y) = ap(by + c) = q_n \prod_{j=1}^{n} (y - y_j),$$

for fixed complex a, b and c such that $ab \neq 0$,

$$\|q(y)\| = 1, \quad 1/4 < r(\widetilde{C}, q(y)) = \max_j |y_j| \le 1, \quad nb\widetilde{C} = \sum_j (x_j - c).$$

Here, several choices of the values of a, b and c are possible. For instance, we may set $b = 4r(C, p(x))$, $c = C + 0.5$ and define a from the equation $\|q(y)\| = 1$. (Actually, in a more careful implementation, we may somewhat relax some constraints of (5) and (6) in order to be able to choose lower precision values for a, b, and c, thus decreasing the Boolean cost of the computations. For instance, we may require that $\frac{1}{2} \le \|q(x)\| \le 1$ (rather than $\|q(x)\| = 1$) and choose a to be a power of 2.)

We are now ready to state our main result, which we will prove in §§6 and 7 (also compare Remark 8).

THEOREM 1. *Suppose that we are given complex coefficients $p_0, \ldots, p_n$ of the polynomial $p(x)$ of* (1), *satisfying* (5), (6), *and four positive α, β, δ and ϵ such that*

$$(7) \qquad 1 > \delta > 0, \quad n|p_n|20^{-n} > \epsilon.$$

Then there exists a randomized algorithm (*of Las Vegas type*) *that either reports a failure with a probability at most δ or, otherwise, computes an incomplete numerical factorization of $p(x)$* (*within error norm ϵ*) *into a product of polynomials, each of degree less than $n/2$. The algorithm supports the following parallel and sequential complexity estimates for this computation*:

$$(8) \qquad c_A = O_A\left(\phi(n, \epsilon) \log n, n^3 \log(1/\delta)/\phi(n, \epsilon)\right),$$

$$(9) \qquad (sc)_A = O_A(\phi(n, \epsilon) n^2 \log n + n(\log n)^2 \log(1/\delta), 1),$$

where $\phi(n, \epsilon) = \log(\log(1/\epsilon) + (n/\epsilon^{1/n}))$.

Remark 1. If

$$(10) \qquad \log(1/\delta) = O(\log n), \qquad \log(1/\epsilon) = O(n^\alpha (\log n)^\beta),$$

then $\log\log(1/\epsilon) = O(\log n)$, $\log(1/\epsilon^{1/n}) = O(n^{\alpha-1}(\log n)^\beta)$, and the estimates (8) and (9) are specified as follows:

$$(11) \qquad c_A = O_A\big(n^{\alpha-1}(\log n)^{\beta+1}, n^3(\log n)^{1-\beta}\big),$$

$$(12) \qquad (sc)_A = O_A(n^{1+\alpha}(\log n)^{\beta+1}, 1).$$

(In the summary, we cited (11) and (12) for $\alpha = \beta = 1$.)

Remark 2. We also show (in §9) how the complexity bounds (8) and (9) can be ensured, except for a narrow class of input polynomials $p(x)$, with $\phi(n, \epsilon)$ replaced by the substantially smaller quantity $\psi(n, \epsilon) = \log(\log(1/\epsilon) + (n/\epsilon^{1/n^2}))$, which (for all but a narrow class of input polynomials $p(x)$) implies the bounds (11) and (12), even provided that the upper bound of (10) on $\log(1/\epsilon)$ is increased by the factor of n, to

$$\log(1/\epsilon) = O(n^{\alpha+1}(\log n)^{\beta}).$$

In particular, for $\alpha = 1$, which implies the time bound $(\log n)^{\beta+1}$ in (11), we have the bound $\log(1/\epsilon) = O(n^2(\log)^{\beta})$ versus $O(n(\log n)^{\beta})$ of (10) for $\alpha = 1$. According to the estimates of [4] (p. 1084, the last paragraph of §2) or [17], (Lemma 4.1 and the paragraph following equation (2.8)) (or alternatively, this can be deduced by analyzing the implications of our Proposition 8), this improvement gives us a basis for separating and isolating the zeros of $p(x)$ from each other and, therefore, for their rapid subsequent refinement via Newton's iteration.

We organize the remaining part of the paper as follows: In §2 we estimate some correlations between the absolute value of the discriminant of $p(x)$ (or of the resultant $R(p, p')$ of $p(x)$ and $p'(x)$) and the mth diameter of the set of zeros of $p(x)$. In §3 we estimate $|R(p + \Delta, p')|$ as a function in Δ. In §4 we show our hybrid deterministic algorithm. In §5 we combine the results of §§2–4 into our basis lemma (Proposition 9). In §6 we summarize our study in the form of a parallel algorithm. In §7 we modify this algorithm for the sequential computation and will deduce the complexity estimates of Theorem 1. In §8 we comment on the recursive application of our algorithms. In §9 we improve our complexity estimates over a large class of the input polynomials $p(x)$. Finally, in the appendix we recall some auxiliary results on perturbation of polynomial zeros.

Hereafter, all logarithms are to the base 2, and each polynomial zero of multiplicity μ is counted as μ zeros.

2. The magnitude of the resultant and the distances between polynomial zeros. In this section we will relate the distances between some zeros of a polynomial $p(x)$ of (1) to the absolute value of R, the resultant of this polynomial and its derivative $p'(x)$. We will make use of the two following expressions for R:

$$(13) \qquad R = R(p_0, \ldots, p_n) = (-1)^{n(n-1)/2} p_n^{2n-1} \prod_{1 \le i < j \le n} (x_i - x_j)^2;$$

R equals the determinant of the $(2n - 1) \times (2n - 1)$ Sylvester (resultant) matrix,

$$(14) \qquad R = \det S\,, \quad S = \begin{bmatrix} p_n & & O & (np_n) & & O \\ \vdots & \ddots & & \vdots & \ddots & \\ p_2 & \cdots & p_n & (2p_2) & & \\ p_1 & \cdots & p_{n-1} & p_1 & \cdots & (np_n) \\ p_0 & \cdots & p_{n-2} & & & \\ & \ddots & \vdots & & \ddots & \vdots \\ O & & p_0 & O & & p_1 \end{bmatrix}.$$

Equation (14) implies, in particular, the following proposition.

PROPOSITION 2. *The resultant R is a polynomial in* $p_0, p_1, \ldots, p_n$ *with integer coefficients, having the total degree* $2n - 1$.

We will recall (4), ignore the case of multiple zeros, assume for a moment that $R \neq 0$, and will immediately deduce from (13) the following proposition.

PROPOSITION 3. *Let us denote that*

$$D = D(p_0, \dots, p_n) = |R(p_0, \dots, p_n)/p_n^{2n-1}|. \tag{15}$$

Then for any integer m, $2 \leq m \leq n$, *we have that*

$$D \leq d_n^{n(n-1)}(d_m/d_n)^{m(m-1)}, \qquad m = 1, 2, \dots, n. \tag{16}$$

Hereafter, we will assume that n is even, $n = 2m$, so that $m(m-1) = n(n-2)/4$. Then we will deduce from (16) that

$$D/d_n^{n(3n-2)/4} \leq d_m^{n(n-2)/4}, \tag{17}$$

$$D^{4/n(n-2)}/d_n^{(3n-2)/(n-2)} \leq d_m. \tag{18}$$

3. Probabilistic lower bound on the magnitude of the resultant. Let Δ denote a random complex parameter under the uniform probability distribution on the circle

$$|\Delta| = \rho, \tag{19}$$

where the value ρ will be fixed later on. We are going to deduce some probabilistic lower bound on $|R_h(\Delta)| = |R(p_0, \dots, p_{h-1}, p_h + \Delta, p_{h+1}, \dots, p_n)|$, $0 \leq h \leq n$ (compare (13)). Recall Proposition 2 and rewrite $R_h(\Delta)$ as a polynomial in Δ,

$$R_h(\Delta) = \sum_{i=0}^{n} r_{h,i}\Delta^i = r_{h,k}\prod_{j=1}^{k}(x - x_{j,h}(\Delta)), \qquad k = k(h) \leq n, \tag{20}$$

where $r_{h,i}$ are polynomials in $p_0, \dots, p_n$ with integer coefficients and have total degrees at most $2n - 1 - i$, $i = 0, 1, \dots, n$. Furthermore,

$$r_{h,1} = \partial R/\partial p_h, \qquad h = 0, 1, \dots, n, \tag{21}$$

$$r_{0,n} = 0, \qquad r_{0,n-1} = (-1)^{n(n-1)/2}(np_n)^n. \tag{22}$$

The next simple result is the basis for our algorithms.

PROPOSITION 4. *For a random* $\Delta = |\rho|$, *under the uniform probability distribution on the circle* (19), *for any pair of nonnegative integers* $h \leq n$, $u \leq n$, *and for any fixed positive* ρ, *we have*

$$|R_h(\Delta)| > 0.5|r_{h,u}|\rho^u \tag{23}$$

with a probability at least $1/(2k(h) + 1) \geq 1/(2n + 1)$, *where* $k(h)$ *is defined by* (20).

Proof. Let j be an integer such that

$$|r_{h,u}|\rho^u \leq |r_{h,j}|\rho^j = \max_{0 \leq i \leq n}(|r_{h,i}|\rho^i), \qquad 0 \leq j \leq n. \tag{24}$$

It is well known from analytic function theory that

$$\int_{|\Delta|=\rho}(R_h(\Delta)/\Delta^{j+1})\,d\Delta = 2\pi\sqrt{-1}\,r_{h,j}.$$

Representing Δ as $\rho \exp(2\pi\sqrt{-1}\,\phi)$, we obtain that

$$(1/\rho^j)\int_0^{2\pi} |R_h(\Delta)|\,d\phi = \int_0^{2\pi} |R_h(\Delta)/\Delta^{j+1}|\,\rho\, d\phi \geq 2\pi |r_{h,j}|,$$

$$\frac{1}{2\pi}\int_0^{2\pi} |R_h(\Delta)|\,d\phi \geq |r_{h,j}|\,\rho^j,$$

so that $|r_{h,j}|\rho^j$ is a lower bound on the average value of $|R_h(\Delta)|$ on the circle (19). To complete the proof, we compare this bound with the following upper bound: $|R_h(\Delta)| \leq \sum_{i=0}^{k(h)} |r_{h,i}|\rho^i \leq (k(h)+1)|r_{h,j}|\rho^j$, then recall (24). □

Hereafter, denote that

$$\tilde{p}_{h,n} = p_n + \Delta \text{ if } h = n; \qquad \tilde{p}_{h,n} = p_n \text{ otherwise.}$$

Assume that $\tilde{p}_{h,n} \neq 0$ for all h. Recall (20) and, similarly to the definitions (15) and (4), denote that

$$D_h(\Delta) = |R_h(\Delta)/\tilde{p}_{h,n}^{2n-1}|,$$

$$d_{h,k}(\Delta) = \min_{|J|=k}\max_{i,j\in J} |x_{h,i}(\Delta) - x_{h,j}(\Delta)|, \qquad k = 2, 3, \ldots, n, \quad h = 0, 1, \ldots, n,$$

and rewrite (18) as

$$(25) \qquad (D_h(\Delta))^{4/n(n-2)} / (d_{h,n}(\Delta))^{(3n-2)/(n-2)} \leq d_{h,m}(\Delta), \qquad m = n/2.$$

Rewrite $(3n-2)/(n-2)$ as $3 + 4/(n-2)$, combine (23) and (25), and obtain that

$$(26) \qquad d_{h,m}(\Delta) > (0.5\,|\,r_{h,u}\tilde{p}_{h,n}^{1-2n}|\rho^u)^{4/n(n-2)} / (d_{h,n}(\Delta))^{3+4/(n-2)},$$

$u, h = 0, 1, \ldots, n$, with a probability at least $1/(2n+1)$.

In particular, substitute (22) into (26), for $h = 0$, $u = n-1$, and arrive at the inequality

$$(27) \qquad d_{0,m}(\Delta) > (0.5n^n)^{4/n(n-2)} |\rho/p_n|^{4(n-1)/n(n-2)} / \left(d_{0,n}(\Delta)\right)^{3+4/(n-2)},$$

where $\lim_{n\to\infty}(0.5n^n)^{4/n(n-2)} = 1$.

Also, substitute (21) into (26), for $u = 1$ and $h = 0, 1, \ldots, n$, to obtain that

$$(28) \qquad d_{h,m}(\Delta) > \left(0.5|\tilde{p}_{h,n}^{1-2n}\frac{\partial R}{\partial p_h}|\rho\right)^{4/n(n-2)} / (d_{h,n}(\Delta))^{3+4/(n-2)}.$$

4. A deterministic algorithm for polynomial factorization. Hereafter, $S(C, r)$ will denote the square on the complex plane, with the center C and with the sides parallel to the real and imaginary axes and having length $2r$. $D(C, r)$ will denote the disc on the complex plane, with the center C and radius r. We assume that $S(C, r)$ and $D(C, r)$ denote closed complex domains.

$\tilde{R}/r$ is called the *isolation ratio* of the square $S(C, r)$ or of the disc $D(C, r)$ if $\tilde{R}$ is the minimum value such that the domains $S(C, \tilde{R}) - S(C, r)$ or, respectively, $D(C, \tilde{R}) - D(C, r)$ contain a zero of $p(x)$.

We next recall some known results.

PROPOSITION 5. *Given the coefficients $p_0, \dots, p_n$, a complex C and a constant θ, $0 < \theta < 1$, it is possible, at the cost $O_A(\log n, n)$, to evaluate r such that*

$$\theta r \le r(C, p(x)) = \max_{1 \le j \le n} |C - x_j| \le r. \tag{29}$$

Remark 3. $d_n \le 2r(C, p(x)) \le 2r$, under the notation of (29).

Proposition 5 relies on an old algorithm of Turan (see [27] and [28]) complemented with more recent estimates for the complexity of its blocks.

PROPOSITION 6 [19]. *Let a complex C and a positive r be two given values that satisfy* (6) *and* (29). *Then there exists a natural $H = H(n, r, \theta) = O(\log n)$ such that, for every $K > H$, it is possible, at the cost $O_A(K \log n, n^2)$, to compute complex C_i, real r_i, and natural $g \ge 2$ and $k_i \ge 1$, $i = 1, \dots, g$, such that for all i*

$$r_i \le rn/2^K, \tag{30}$$

and $S(C_i, r_i)$ has an isolation ratio of at least 3 *and contains exactly k_i zeros of $p(x)$; furthermore, $\sum_{i=1}^{g} k_i = n$, and $S(C_j, r_j) \cap S(C_h, r_h)$ is empty for any pair of j and h such that $1 \le j < h \le g$.*

The proof of Proposition 6 in [19] exploits and improves the search-and-exclusion construction of Weyl for approximating polynomial zeros (compare [12], [22], and [24] on other applications of Weyl's construction).

PROPOSITION 7. *Let $S(C_i, r_i)$, the squares defined in Proposition* 6, *for $i = 1, \dots, g$, be given to us, together with the associated integers k_i and with a positive ϵ^*. Let us denote*

$$b = b(\epsilon^*) = (1/n)\log(1/\epsilon^*). \tag{31}$$

Then, at the cost $O_A(\log n \log(bn), gn \log b/\log(bn))$, the coefficients of $g \le n$ monic polynomials $p_1(x), \dots, p_g(x)$ can be computed such that the polynomial $p_i(x)$ has exactly k_i zeros, all lying in $S(C_i, r_i)$, $i = 1, \dots, g$, and

$$\|p(x) - p_1(x) \cdots p_g(x)\| \le \epsilon^*. \tag{32}$$

Proposition 7 follows from the results of [25] (Corollary 4.3 and §12) applied to the discs $D(C_i, r_i\sqrt{2})$, $i = 1, \dots, g$; such a disc $D(C_i, r_i\sqrt{2})$ contains exactly k_i zeros of $p(x)$ (lying in the square $S(C_i, r_i)$) and has an isolation ratio at least $3/\sqrt{2} > 2$ (compare Remark 4).

Combining Propositions 5–7, we devise the following algorithm.

Algorithm 1. Input: the complex coefficients $p_0, \dots, p_{n-1}$ of a polynomial $p(x)$ of (1) and a positive ϵ^*.

Output: positive integers $g > 1$, $k_1, \dots, k_g$ (such that $k_1 + \cdots + k_g = n$) and the coefficients of monic polynomials $p_i(x)$ of degrees k_i, $i = 1, \dots, g$, satisfying (32).

Stage 1. Compute C and r satisfying (6) and (29).

Stage 2. Compute the squares $S(C_i, r_i)$ defined in Proposition 6.

Stage 3. Compute a factorization (32), by following Proposition 7.

We immediately deduce the bounds

$$C_A^* = O_A((K + \log(bn))\log n, n^2) \tag{33}$$

on the parallel cost of the computation by Algorithm 1, where b has been defined in Proposition 7 and where we may choose any K exceeding $H = O(\log n)$. (Later on, we will see some advantages of choosing a larger value of K.)

We may recursively apply Algorithm 1 in whose input set we replace $p(x)$ by the polynomials $p_1(x), \dots, p_g(x)$, as long as they have degrees exceeding 1 (these g applications can be performed concurrently, and similarly in the further recursive steps). The maximum degree of the output polynomials of Algorithm 1 is $n - g + 1$ or less, so that in at most $n - 1$ its applications and at the overall cost bounded by $O_A((K + \log(bn))n \log n, n^2)$, we will arrive at the factorization (2) of $p(x)$ into linear factors. Here, we impose the same error bound ϵ^* for the numerical factorization of all the auxiliary polynomials that we factorize in this recursive process. We choose this bound ϵ^* sufficiently small, so as to ensure the error bound ϵ for the complete numerical factorization (2). Specifically, we are guided by the next proposition from §5 of [25], which relates ϵ^* to ϵ:

PROPOSITION 8. *Let*

$$\|p(x) - p_1(x) \cdots p_h(x)\| \le \epsilon h/n, \quad \|p_1(x) - f(x)g(x)\| \le \epsilon^*,$$

$$\epsilon^* = \frac{\epsilon \|p_1(x)\|}{n2^n \|p(x)\|}.$$

Then

$$\|p(x) - f(x)g(x)p_2(x) \cdots p_h(x)\| \le (h + 1)\epsilon/n.$$

Due to Proposition 8, it is sufficient to choose

$$\epsilon^* = \frac{\epsilon \|w(x)\|}{n2^n \|p(x)\|} \ge \frac{\epsilon}{n2^n \|p(x)\|} \tag{34}$$

whenever Algorithm 1 is applied to a monic polynomial $w(x)$ in the recursive process described above.

Next suppose that, by modifying the first step of this recursive process, we may ensure a more rapid decrease of the degrees of the factors, such that

$$\max_{1 \le i \le g} \deg p_i(x) < 0.5 \deg p(x) = n/2,$$

and similarly at all the next recursive steps. Then it would have sufficed to use $\lfloor \log n \rfloor$ (rather than $n - 1$) concurrent recursive steps of application of Algorithm 1 (as before, we ensure the desired output error bound ϵ by imposing the error bound ϵ^* of (34) at all the recursive steps).

We easily estimate that in this case the complexity of recursive step i is $O_A((K + \log(bn)) \log n, n^2/2^i)$, for $i = 1, 2, \dots, \lfloor \log n \rfloor$, which implies that the overall complexity of the complete numerical factorization of $p(x)$ into linear factors is $O_A((K + \log(bn)) \log^2 n, n^2/\log n)$. (To arrive at this bound, we apply Brent's principle of [10] by slowing down those recursive steps that otherwise would have used more than $n^2/\log n$ processors.)

In comparison to our previous bound of $O_A((K + \log(bn))n \log n, n^2)$ in the case of unbalanced recursive splitting of $p(x)$, we now note the decrease of both the sequential time bound (by the factor of n) and the parallel time bound (by the factor of $n/\log n$), so that Algorithm 1 performs much more effectively in the cases where the factors computed at its recursive steps have smaller degrees. In the next sections we will apply a randomization technique to split $p(x)$ into factors of degrees less than $n/2$.

Remark 4. Algorithm 1 is a modification (based on the results of [19]) of the algorithm of [25]. The recursive step of [25] splits $p(x)$ into two factors, whose zeros lie inside and, respectively, outside a fixed disc that has an isolation ratio of the order of $1 + 1/n$. Since we deal with the squares $S(C_i, r_i)$ of Propositions 6 and 7, having isolation ratios at least 3, the computation of the splitting by using Algorithm 1 is substantially simpler than the similar computation in [25].

5. How to ensure a balanced partition of the set of zeros. In the next result, we use D and r defined by (15) and (29).

PROPOSITION 9. *Let the integer K of Proposition 6 be chosen such that*

$$2^{K-0.5} > n(2r)^{4(n-1)/(n-2)} \,/\, D^{4/(n(n-2))}. \tag{35}$$

Then $2\sqrt{2}r_i < d_m$ for $i = 1, \dots, g$, that is, every square $S(C_i, r_i)$, $i = 1, \dots, g$, of Proposition 6 contains less than $m = n/2$ zeros of $p(x)$.

Proof. Proposition 9 immediately follows from the comparison of the upper bound (30) on r_i with the lower bound (18) on d_m. (Compare also Remark 3 and observe that the diameter of the square $S(C_i, r_i)$ equals $2\sqrt{2}\,r_i$.) □

Remark 5. The reader may extend (18) and Proposition 9, as well as our subsequent study, by choosing a smaller m, say, $m = \sqrt{n}$.

Proposition 9 can be immediately extended to the case where $p(x)$ is replaced by $p(x)+\Delta$, for Δ of (19), D by $D_0(\Delta)$ (see §3) and similarly r of (29) by $r_0(\Delta)$; in particular, we may rewrite the assumption (35) as follows:

$$D_0(\Delta) > (2r_0(\Delta))^{(n-1)n}\, n^{(n-2)n/4}\, 2^{-(K-0.5)(n-2)n/4}, \tag{36}$$

where the quantity K will be estimated later on.

Now let d, K, Δ and $\rho = |\Delta|$ be such that $d \geq 2r_0(\Delta)$,

$$0.5|r_{0,n-1}\, p_n^{1-2n}|\rho^{n-1} \geq d^{(n-1)n}\, n^{n(n-2)/4} \,/\, 2^{(K-0.5)n(n-2)/4}. \tag{37}$$

Apply Proposition 4 for $h = 0$, $u = n - 1$ and deduce that (with a probability at least $1/(2n + 1)$) we have:

$$\begin{aligned} D_0(\Delta) = |R_0(\Delta) \,/\, p_n^{2n-1}| &> 0.5|r_{0,n-1} \,/\, p_n^{2n-1}|\rho^{n-1} \\ &\geq d^{(n-1)n}\, n^{n(n-2)/4}\, 2^{-(K-0.5)n(n-2)/4}. \end{aligned}$$

Since $d \geq 2r_0(\Delta)$, the latter inequality implies (36), and, consequently, implies the extension of Proposition 9 to the case where the polynomial $p(x) + \Delta$ replaces $p(x)$.

6. The factorization algorithm. Now we are ready to summarize our previous study, by devising the following randomized (of Las Vegas type) algorithm.

Algorithm 2. Input: the complex coefficients $p_0, \dots, p_{n-1}$ of a polynomial $p(x)$ of (1), a positive ρ such that

$$r(C, p(x)) \leq 2r(C, p(x) + \Delta) \leq 3r(C, p(x)) \text{ if } |\Delta| = \rho, \quad C = -p_{n-1}/(np_n) \tag{38}$$

(compare Remark 6), and a tolerance δ to the probability of a failure, $0 < \delta < 1$ (compare (7)).

Output: FAILURE with a probability at most δ or, otherwise, an integer g, $g > 2$, a complex value Δ, $|\Delta| = \rho$, and the coefficients of g polynomials $p_i^*(x, \Delta)$ having degrees less than $n/2$ and satisfying the inequality (40) below.

Stage 1. Successively compute the values

$$\nu = \lceil (2n + 1)\log(1/\delta) \,/\, \log 3 \rceil, \tag{39}$$

then C and r satisfying (6) and (29), setting throughout $\Theta = 0.99$ (compare (29) and Remark 6), then $d = 3.1\,r$, and finally, the minimum positive integer K exceeding H of Proposition 6 and satisfying (37). Choose ν independent random values Δ on the circle $|\Delta| = \rho$, under the uniform probability distribution.

Stage 2. For all such values Δ and for $\Delta = 0$, concurrently compute $D_0(\Delta)$. If (36) has been satisfied for none of the values Δ, end this computation and output FAILURE. Otherwise, fix some Δ for which (36) holds, apply Algorithm 1 to the polynomial $p(x) + \Delta$ in order to compute and to output a natural $g > 2$ and the coefficients of the polynomials $p_i^*(x, \Delta)$ that approximate g distinct nonconstant factors $p_i(x, \Delta)$ of $p(x) + \Delta$, for $i = 1, \ldots, g$, such that

$$(40) \qquad \begin{gathered} p(x) + \Delta = \textstyle\prod_{i=1}^{g} p_i(x, \Delta); \quad \|p(x) + \Delta - \prod_{i=1}^{g} p_i^*(x, \Delta)\| \leq \rho; \\ \|p(x) - \textstyle\prod_{i=1}^{g} p_i^*(x, \Delta)\| \leq 2\rho, \end{gathered}$$

and end the computations. (Note that 2ρ plays the role of ϵ^* of (32), (34), and Algorithm 1.)

Let us next show the correctness of the algorithm. First observe that $d = 3.1\,r \geq 3.1\,r\big(C, p(x)\big)$ (due to (29)). Recall (38) and obtain that $d \geq 2.05\,r(C, p(x) + \Delta)$. Extend (29) by replacing $p(x)$ by $p(x) + \Delta$ and r by $r_0(\Delta)$, and obtain that $d \geq 2.05\,\Theta\, r_0(\Delta)$. Substitute $\Theta = 0.99$ and obtain that

$$(41) \qquad d > 2r_0(\Delta).$$

Now apply Proposition 4, for $h = 0$, $u = n - 1$, and deduce that, with a probability at least $1/(2n + 1)$,

$$|R_0(\Delta)| > 0.5 r_{0,n-1}\rho^{n-1},$$

which implies that

$$D_0(\Delta) = |R_0(\Delta) \,/\, p_n^{2n-1}| > 0.5\, r_{0,n-1}|p_n^{1-2n}|\rho^{n-1}.$$

The latter inequality implies (36) since we have chosen the value of K satisfying (37). Therefore, we may apply Proposition 9 extended to the case of the input polynomial $p(x) + \Delta$. Then, application of Algorithm 1 at stage 2 of Algorithm 2 gives us a numerical factorization of $p(x) + \Delta$ satisfying (40), and we observe that the probability of a failure in an application of stage 2 to $p(x) + \Delta$ for all the ν values $\Delta \neq 0$ is at most $(1 - 1/(2n+1))^{\nu} < 1/3^{\log(1/\delta)/\log 3} = \delta$ (see (39)).

Remark 6. The assumption (38) surely holds for any sufficiently small positive ρ. In particular, we may deduce from Proposition 11 of Appendix A that it is sufficient if

$$(42) \qquad 2\rho < 40^{-n}|p_n|\,.$$

Of course, the reader may replace the "magic" numbers 2 and 3 in (38) and the value $\Theta = 0.99$ by various other candidate values, so as to preserve the correctness of Algorithm 2, respectively modifying (42).

Remark 7. In a heuristic attempt of derandomization, we may further modify stage 2 of Algorithm 2 by first computing the values of $R_0(\Delta)$ for $\Delta = \rho \exp(2\pi\sqrt{-1}\,k/n)$, $k = 0, \ldots, n - 1$, then the coefficients $r_{0,0}, \ldots, r_{0,n-1}$ of $R_0(\Delta)$ (by first performing FFT (which gives us $r_{0,i}\rho^i$, for $i = 0, 1, \ldots, n - 1$) and then scaling) and, finally, a complex Δ, $|\Delta| = \rho$, for which $|R_0(\Delta)|$ reaches its maximum on the circle $|\Delta| = \rho$. We may approximate (within the factor $\sqrt{2}$) this maximum by the maximum of the absolute value of its real and imaginary parts. If the coefficients $p_0, \ldots, p_n$ and, therefore, $r_{0,0}, \ldots, r_{0,n-1}$ are real, then these real and imaginary parts are represented as the trigonometric polynomials $\sum_{k=0}^{n-1} \rho^k r_{0,k} \cos(kt)$ and $\sqrt{-1}\sum_{k=0}^{n-1} \rho^k r_{0,k} \sin(kt)$, respectively, where $\Delta = \rho \exp(t\sqrt{-1})$ and t ranges from 0 to 2π. For a large class of input polynomials $p(x)$, such a reduction to approximating the maximum absolute value of the above trigonometric polynomials simplifies the original task of the factorization of a polynomial.

7. Computational complexity estimates. Next, we will assume that $p(x)$ and x have been scaled so that

$$d \leq 1 \tag{43}$$

and will estimate the parallel complexity c_A of performing Algorithm 2. Clearly, c_A is dominated by the complexity of computing $D_0(\Delta)$ for all the ν values Δ (bounded by $O_A(\log^2 n, \nu n^2 / \log n)$, due to [21]) and of applying Algorithm 1 (see (33)), so

$$c_A = O_A\big[(K + \log(bn)) \log n, n^2(1 + \nu/(K + \log(bn)))\big], \tag{44}$$

where K denotes the minimum integer exceeding H (defined in Proposition 6) and satisfying (37), $bn = -1 + \log(1/\rho)$ (compare (31)) and ν is defined by (39).

Now observe that (40) implies (32), for any $\rho \leq \epsilon^*/2$ and for $p_i(x) = p_i^*(x, \Delta)$, $i = 1, \dots, g$. Suppose that α, β, δ and ϵ are given as in Theorem 1, such that (7) holds. Then (38) holds for $\rho = \epsilon^*/2$ (see (5), (34), and Remark 6). Set $\rho = \epsilon^*/2$ and apply Algorithm 2.

Next, recall the definition of K given at stage 1 of this algorithm and choose the minimum K satisfying this definition, that is, the minimum $K \geq H + 1$ satisfying (37). Formally, let

$$K = \max\{H + 1, \lceil K^* \rceil\}, \tag{45}$$

where K^* is such that setting $K = K^*$ turns (37) into an equation, that is,

$$2^{(K^*-0.5)n(n-2)/4} = 2d^{(n-1)n}\, n^{(n-2)n/4}\, |p_n^{2n-1} / r_{0,n-1}\rho^{n-1}|.$$

Substitute (22), take logarithms on both sides, divide the resulting equation by $n(n-2)/4$ and obtain that

$$\begin{aligned} K^* &= 0.5 + 4/(n-2)n \\ &\quad + (1 - 4/(n-2)) \log n + ((4n-4) / (n-2)n) \log |p_n d^n / \rho|. \end{aligned} \tag{46}$$

Replace ρ by $\epsilon^*/2$, recall (5), (34), (43) and Proposition 9, and deduce from (43), (46) and the equation $H = O(\log n)$ (see Proposition 6) that

$$K = O(\log n + (1/n) \log(1/\epsilon)) = O(\log(n/\epsilon^{1/n})). \tag{47}$$

We also have that

$$bn = O(\log(1/\epsilon^*)), \tag{48}$$

$$\nu = O(n \log(1/\delta)) \tag{49}$$

(compare (31), (39)). Combining (34), (44)–(49), we obtain that $K + \log(bn) = \phi(n, \epsilon) = O(\log(\log(1/\epsilon) + (n/\epsilon^{1/n})))$, and arrive at (8):

$$c_A = O_A(\phi(n, \epsilon) \log n, n^3 \log(1/\delta) / \phi(n, \epsilon)).$$

To deduce the bound (9), we modify Algorithm 2 as follows.

Algorithm 3.

(a) Compute $R_0(\Delta)$ for n values Δ equal to all the nth roots of 1; for each value of Δ, such an evaluation amounts to computing the determinant of a Sylvester matrix, at the sequential cost $O_A(n(\log n)^2, 1)$ [7], [8], [16]; this means $O_A((n \log n)^2, 1)$ for all the values Δ;

(b) compute the coefficients of $R_0(\Delta)$ as a polynomial in Δ (see (14) and (20)) by means of the inverse FFT, at the cost $O_A(\log n, n)$ or $O_A(n \log n, 1)$ [1];

(c) define ν by (39) and compute $D_0(\Delta)$, for ν random and independent values Δ on the circle $|\Delta| = \rho$, at the overall cost $O_A((\nu + n)\log^2 n, 1)$ [1], [7], [9];

(d) among these ν values, choose the value Δ that maximizes $|D_0(\Delta)|$ and output FAILURE if (36) does not hold, for this value of Δ;

(e) otherwise, if (36) holds, apply Algorithm 1 to the polynomial $p(x) + \Delta$, at the cost $C_A^* = O_A((K + \log(bn))\log n, n^2)$, bounded according to (33), (45)–(48); by Brent's principle of [10], this implies the sequential cost bound $(sC)_A^* = O_A(n^2(K + \log(bn))\log n, 1)$.

Summarizing all these estimates for the complexity of performing Algorithm 3 and assuming (7), (38), and (39), we arrive at the bound (9) on the sequential complexity of computing the approximate factorization (2), thus proving Theorem 1. □

Remark 8. We may perform stage (a) of Algorithm 3 at the cost $O_A(\log^2 n, n^3/\log n)$ by concurrently applying the parallel algorithm of [21] for all the n values of Δ. Stage (c) can be performed at the parallel cost $O_A(\log^2 n, (\nu + n)\log\log(\nu + n))$ [7], which turns into $O_A(\log^2 n, n\log(1/\delta)\log\log(n\log(1/\delta)))$ under (39) and which is dominated by the above bound on the cost at stage (a) under the mild condition that $\log(1/\delta) = o(n^2/\log n)$. Assuming this condition and recalling the bound $O_A(\phi(n,\epsilon)\log n, n^2)$, $\phi(n,\epsilon) = O(\log(\log(1/\epsilon) + (n/\epsilon^{1/n})))$, on the cost at stage (e), we obtain the overall cost bound

$$O_A(\phi(n,\epsilon)\log n\,,\; n^2 + n^3/\phi(n,\epsilon)), \tag{50}$$

which slightly improves (8).

8. Extension to recursive factorization. Unless FAILURE has been output, stages 1 and 2 of Algorithms 2 or 3 can be repeated with each of the polynomials $p_i(x, \Delta)$ of degree $k_i > 1$ replacing $p(x)$, with k_i replacing n in (37), and with

$$\nu_i = \lceil (2k_i - 1)\log(1/\delta)\,/\,\log 3 \rceil$$

replacing ν of (39). Then the failure probability is at most δ at this factorization stage for all the $p_i^*(x, \Delta)$. The difficulty with this extension of Algorithms 2 and 3 is that in order to compute the factorization of $p_i^*(x, \Delta)$, we need to replace n by k_i in both (37) and (46), so that at this step, our estimate gives us $K = O((1/k_i)\log(1/\rho))$, rather than $K = O((1/n)\log(1/\rho))$. At the subsequent recursive steps, we need to replace k_i, in this expression for K, by smaller and smaller values. This means that we need to choose larger and larger values of K to ensure the desired bound on the approximation error of the factorization. This would increase the running time of the algorithm. Thus, at some recursive step, the estimated time of our randomized computation may exceed the estimated time bound for the recursive step of the deterministic algorithm (based on Algorithm 1), to which we shall shift at this point. It is desired, of course, to decrease the error of the factorization, so as to take advantage of the randomization as long as possible, reaching the factors of $p(x)$ that have smaller degrees. To achieve this, we will next slightly modify our randomization approach. (An alternative is to rely on Remark 5 instead of recursion.)

9. Decreasing upper bounds on the factorization error. Algorithms 2 and 3 and their analysis in §7 relied on (22) and (27). For a very large class of input polynomials $p(x)$, we may substantially improve the asymptotic complexity bounds (8) and (9). In particular, we may insure the complexity estimates (11) and (12) under the assumption on ϵ that is weaker than (10), namely, if we only assume that, roughly,

$$\log(1/\epsilon) = O(n^{1+\alpha}(\log n)^{\beta}), \tag{51}$$

that is, for a large class of inputs, we may arrive at a superior approximation to the factorization of $p(x)$ by using roughly the same amount of computational resources. (We refer to (55) and (56) below, for a more precise statement of the assumption (51).) The improvement is important in the recursive application of the factorization algorithms (see the previous section).

Let us supply some details. Denote

$$w_h = 0.5|\tilde{p}_{h,n}^{1-2n}\,\partial R/\partial p_h|, \qquad h = 0, 1, \dots, n, \tag{52}$$

substitute this expression on the right side of (28), also replacing $d_{h,n}(\Delta)$ by d of (41), $d \geq d_{h,n}(\Delta)$, and obtain that

$$d_{h,m}(\Delta) > (w_h\rho)^{4/n(n-2)} \,/\, d^{3+4/(n-2)}. \tag{53}$$

We need to choose K such that $dn\sqrt{2}/2^K < d_{h,m}(\Delta)$, due to Proposition 6. Assume that $w_h \neq 0$ and extend Proposition 9 replacing (45) and (46) by the following expression:

$$K = \max\left\{H+1, \left\lceil 0.5 - \frac{4}{n(n-2)}\log(w_h\rho) + \log n + \frac{4n-4}{n-2}\log d\right\rceil\right\}. \tag{54}$$

Assume that $\rho = \epsilon^*/2$ and $d \leq 1$ (compare (34) and (43)), and deduce that

$$K = O(\log n + (1/n^2)\log(1/(w_h\epsilon))) = O(\log n \,/\, (w_h\epsilon)^{1/n^2}), \text{ unless } w_h = 0.$$

This is a considerable improvement of (47) unless w_h is small. Clearly, w_h does not decrease as $\epsilon \to 0$, and (52) and Proposition 2 imply that

$$w_h \geq 0.5\, z\, |z\tilde{p}_{h,n}|^{1-2n}$$

if $z \neq 0$, zp_i are integers for all i and $\partial R/\partial p_h \neq 0$. Furthermore, we may evaluate $\partial R/\partial p_h$, for $h = 0, 1, \dots, n$, at the same asymptotic computational cost (bounded by $O_A(\log^2 n, n^2 \,/\, \log n)$ or alternatively by $O_A(n\log^2 n, 1)$) that we need for the evaluation of R. We achieve this just by applying to R the parallel algorithm of [13], which extends the sequential algorithm of [3] and [15]. (The algorithm of [13] computes all the first order partial derivatives of any polynomial $p(y_1, \dots, y_s)$ at the cost $O_A(t, p)$, provided that an algorithm is available that computes $p(y_1, \dots, y_s)$ at the cost $O_A(t, p)$, by using only arithmetic operations, with no branchings.)

Thus, initially, we may compute $\partial R/\partial p_h$ and $w_h\rho$, for $h = 0, 1, \dots, n$, then choose h for which w_h is maximum, and finally compare $w_h\rho$ with $0.5\,n^n\,|\rho \,/\, p_n|^{n-1}$ (see (22), (23)). If the latter value is greater, we shall go to Algorithms 2 or 3 with no changes. Otherwise, we shall apply one of these algorithms, replacing $D_0(\Delta)$ by $D_h(\Delta)$ and K of (45) and (46) by K of (54).

In the latter case, we shall replace $\phi(n, \epsilon)$ in (8), (9), and (50) by $\psi(n, \epsilon) = \log(\log(1/\epsilon) + n/(\epsilon w_h)^{1/n^2})$, and thus shall arrive at the improved overall complexity estimates. In particular, we may deduce (11) and (12) relaxing our previous assumption that $\log(1/\epsilon) = O(n^\alpha(\log n)^\beta)$ and only assuming that

$$\log\log(1/\epsilon) = O((\log n)^\beta), \tag{55}$$

$$\log(1/(\epsilon w_h)) = O(n^{\alpha+1}(\log n)^\beta), \quad \text{for fixed} \quad \alpha \geq 1,\ \beta \geq 1. \tag{56}$$

(Equations (55) and (56) turn into (51) if, say, $1/w_h = O(1)$.)

Remark 9. Instead of computing all the partial derivatives, we may just try to factorize the polynomial $\sum_{i=0}^{n}(p_i + c_i\Delta)x^i$ where c_i have been chosen at random, say, in the real interval from 1 to 2 and Δ has been chosen as before, at random on the circle (19) for an appropriate ρ. This approach (for appropriate ρ) shall work as long as $\partial^h(\det S)/(\partial p_i)^h \neq 0$ for some $h = O(1)$ and for some $i \geq 1$ (not necessarily for $i = 1$).

Appendix A. Perturbation of polynomial zeros. Approximation to polynomial zeros and numerical polynomial factorization involve some nontrivial estimates of how much the zeros are perturbed by the perturbation of the coefficients (compare (2), (3), and Proposition 8). This problem is closely related to our study, and next, for the reader's convenience, we will recall and adjust to our purpose some known results on this topic.

We will first recall a simple corollary (see [12] and [17]) from a perturbation theorem by Ostrowski. This theorem bounds the magnitude of the perturbation of the coefficients so that the magnitude of the resulting perturbation of the zeros of $p(x)$ never exceeds $2^{-\mu}$.

Let us denote that

$$\|u_i x^i\|_\infty = \max_i |u_i|.$$

PROPOSITION 10. *For two monic polynomials $p(x)$ and $p^*(x)$ of degree n, where $\|p(x)\|_\infty < 2^m$, $\|p(x) - p^*(x)\|_\infty \leq 2^{-h}$, the zeros $x_1, \dots, x_n$ of $p(x)$ and $x_1^*, \dots, x_n^*$ of $p^*(x)$ can be enumerated so that* $\max_j |x_j - x_j^*| < 2^{m+\log n+2-h/n}$.

We may improve the latter estimates for a large class of polynomials whose zeros inside and outside the unit circle $|x| \leq 1$ are isolated from each other (see [25]). Alternatively, without imposing the latter restriction, we may just scale x so as to place all the zeros inside this circle (as we have assumed in (6)), then argue somewhat similarly to [25], and arrive at the next improved estimates.

PROPOSITION 11. *Given a natural N, a positive $\epsilon < 0.5/40^N$, and the complex coefficients of two monic polynomials of degree N, $P(x)$ and $P^*(x)$, such that $P(x) = \prod_{j=1}^N (x - x_j)$,*

$$\|P(x) - P^*(x)\| < \epsilon, \tag{57}$$

$$\max_j |x_j| \leq 1, \tag{58}$$

we may enumerate the zeros of $P^*(x)$ so that $P^*(x) = \prod_{j=1}^N (x - x_j^*)$ and

$$|x_j - x_j^*| < 4.4(2\epsilon)^{1/N}, \qquad j = 1, \dots, N. \tag{59}$$

Proof. First observe that $P^*(x) = 0$ implies that

$$|P(x)| = |P(x) - P^*(x)| \leq \|P(x) - P^*(x)\| \max(1, |x|^N). \tag{60}$$

Therefore, the open set

$$T = \{z\colon\ |P(z)| < \epsilon \max\{1, |z|^N\}\}$$

contains all the zeros of $P(x)$ and $P^*(x)$. Furthermore, the homotopy argument shows that in a fixed component T_k of T, the polynomial $P(t, x) = P(x) + t(P^*(x) - P(x))$ has the same number of zeros for all t from 0 to 1; in particular, the two polynomials $P(x) = P(0, x)$ and $P^*(x) = P(1, x)$ have the same number of zeros in T_k.

Now, let $P(x) = 0$, for some $x \in T_k$, so that $|x| \leq 1$ (see (58)). Let

$$\operatorname{arc}(x, y) = \{x(u) = x + u\,z(u)\,,\ |z(u)| = 1,\ 0 \leq u \leq d\} \tag{61}$$

be an arc in T_k with the endpoints $x = x(0)$, $y = x(d)$. Then $|P(x + u\,z(u))| = \prod_j |x - x_j + u\,z(u)|$. On the other hand, by applying first (60) and then (57) and (61), we deduce that $|P(x + u\,z(u))| < \epsilon \max\{1, |x + u\,z(u)|^N\} \leq \epsilon(1 + u)^N \leq \epsilon(1 + d)^N$. Therefore,

$$|s(u)| = \left|\prod_{j=1}^N (u - |x - x_j|)\right| \leq \prod_{J=1}^N |x - x_j + u\,z(u)| < \epsilon(1 + d)^N \quad \text{for} \quad 0 \leq u \leq d.$$

The monic polynomial $s(u)$ has degree N, so that $2\max_{0\leq u\leq d}|s(u)| \geq (d/4)^N$ [11, p. 241]. Consequently, $(d/4)^N < 2\epsilon(1+d)^N, d < 4(1+d)(2\epsilon)^{1/N}$. It follows that

$$d \leq 0.1(1+d),\ d \leq 1/9 \quad \text{for} \quad 2\epsilon \leq 1/40^N,$$

and therefore

$$d < 4.4(2\epsilon)^{1/N},$$

which implies (59). □

Let us now apply Proposition 11 under (42), to ensure (38). Equations (42) and (59) together imply that

$$|r(C, p(x)+\Delta) - r(C, p(x))| < 4.4(2\rho/|p_n|)^{1/n} < 0.11,$$

which together with (6) implies (38). (Note that the alternative application of Proposition 10 (instead of Proposition 11) would require that one replace (42) by a stronger bound, of the order of $\rho = (1/n)O(n)$, in order to deduce (38).)

Acknowledgments. I wish to thank Joseph Lewittes and John Reif for their stimulating discussions, the referees for their helpful comments, also C. Engle for her assistance in typing the manuscript.

Note added in proof. The approach of the author's paper [19] of 1987 has been recently extended and modified in the papers *New techniques for approximating complex polynomial zeros*, Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms (1994), pp. 260–270, and *Arithmetic complexity of approximating complex polynomial zeros*, manuscript, 1994.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1976.

[2] A. V. Aho, K. Steiglitz, and J. D. Ullman, *Evaluating polynomials at fixed set of points*, SIAM J. Comput., 4 (1975), pp. 533–539.

[3] W. Baur and V. Strassen, *On the complexity of partial derivatives*, Theoretical Comp. Sci., 22 (1983), pp. 317–330.

[4] M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, *A fast parallel algorithm for determining all roots of a polynomial with real roots*, SIAM J. Comput., 17 (1988), pp. 1081–1092.

[5] M. Ben-Or and P. Tiwari, *Simple algorithm for approximating all roots of a polynomial with real roots*, J. of Complexity, 6 (1990), pp. 417–442.

[6] D. Bini and V. Pan, *Parallel Complexity of Tridiagonal Symmetric Eigenvalue Problem*, Proc. 2nd Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM Press and Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991, pp. 384–393.

[7] ———, *Numerical and Algebraic Computations with Matrices and Polynomials*, vols. 1 and 2, Birkhäuser, Boston, 1994.

[8] R. R. Bitmead and B. D. O. Anderson, *Asymptotically fast solution of Toeplitz and related systems of linear equations*, Linear Algebra and Its Applic., 34 (1980), pp. 103–116.

[9] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York, 1975.

[10] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. ACM, 21 (1974), pp. 201–206.

[11] S. D. Conte and C. deBoor, *Elementary Numerical Analysis*, McGraw-Hill, New York, 1980.

[12] P. Henrici, *Applied and Computational Complex Analysis*, Wiley, 1974.

[13] E. Kaltofen and M. Singer, Size efficient parallel algebraic circuits for partial derivatives, Tech. Report 90-32, Compt. Sci. Dept., Rensselaer Polytechnic Institute, Troy, New York, 1990.

[14] R. M. Karp and V. Ramachandran, *A survey of parallel algorithms for shared memory machines*, in Handbook Theor. Computer Science, North-Holland, Amsterdam, 1990.

[15] S. Linnainmaa, *Taylor expansion of the accumulated rounding errors*, BIT, 16 (1976), pp. 146–160.

[16] B. R. Musicus, *Levinson and fast Choleski algorithms for Toeplitz and almost Toeplitz matrices*, Internal Report, Lab. of Electronics, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[17] C. A. Neff, *Specified precision polynomial root isolation is in* NC, Proc. 31st Annual IEEE Symp. FOCS, IEEE Computer Science Press, 1990, pp. 152–162, and J. Comp. System Sci., to appear.

[18] V. Pan, *Complexity of parallel matrix computations*, Theoretical Comp. Sci., 54 (1987), pp. 65–85.

[19] ———, *Sequential and parallel complexity of approximate evaluation of polynomial zeros*, Computers and Math. (with Applic.) 14 (1987), pp. 591–622.

[20] ———, *Fast and efficient parallel evaluation of the zeros of a polynomial having only real zeros*, Computers & Math. (with Applications), 17 (1989), pp. 1475–1480.

[21] ———, *Parametrization of Newton's iteration for computations with structured matrices and applications*, Computers and Math. (with Applics.), 24 (1992), pp. 61–75.

[22] ———, *Accelerated Solution of the Symmetric Tridiagonal Eigenvalue Problem*, Tech. Rep. TR 92-6, Computer Sci. Dept., State University of New York, Albany, New York, 1992.

[23] M. Rabin, *Probabilistic algorithms in finite fields*, SIAM J. Comput., 9 (1980), pp. 273–280.

[24] J. Renegar, *On the worst-case arithmetic complexity of approximating zeros of polynomials*, J. of Complexity, 3 (1987), pp. 90–113.

[25] A. Schönhage, *The fundamental theorem of algebra in terms of computational complexity*, manuscript, Dept. of Math., U. of Tübingen, Tübingen, Germany, 1982.

[26] J. T. Schwartz, *Fast probabilistic algorithms for verification of polynomial identities*, J. of ACM, 27 (1980), pp. 701–717.

[27] P. Turan, *Power sum method and approximative solution of algebraic equations*, Math. Comp., 29 (1975), pp. 311–318.

[28] ———, *On a New Method of Analysis and Its Applications*, Wiley & Sons, New York, 1984.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 951–965, October 1994

RANDOMIZED ALGORITHMS FOR MULTIPROCESSOR PAGE MIGRATION*

JEFFERY WESTBROOK†

Abstract. The *page migration* problem is to manage a globally addressed shared memory in a multiprocessor system. Each physical page of memory is located at a given processor, and memory references to that page by other processors incur a cost proportional to the network distance. At times the page may migrate between processors at cost proportional to the distance times D, a page size factor. The problem is to schedule movements on-line so that the total cost of memory references is within a constant factor c of the best off-line schedule. An algorithm that does so is called c-competitive. Black and Sleator gave 3-competitive deterministic on-line algorithms for uniform networks (complete graphs with unit edge lengths) and for trees with arbitrary edge lengths. No good deterministic algorithm is known for general networks with arbitrary edge lengths.

Randomized algorithms are presented for the migration problem that are both simple and better than 3-competitive against an oblivious adversary. An algorithm for uniform graphs is given. It is approximately 2.28-competitive as D grows large. A second, more powerful algorithm that works on graphs with arbitrary edge distances is also given. This algorithm is approximately 2.62-competitive (or, 1 plus the golden ratio) for large D. Both these algorithms use random bits only during an initialization phase, and from then on run deterministically. The competitiveness of a very simple coin-flipping algorithm is also examined.

Key words. on-line algorithms, page migration, competitive analysis, multiprocessors, memory management

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

A common design for a shared memory multiprocessor system is a network of processors, each of which has its own local memory [9], [14], [19]. In such a design, a programming abstraction of a single global memory address space is supported by a virtual memory system that distributes one or more copies of each physical page of memory among the processors' local memories. When processor p wishes to read or write to memory address a, located in page b, it first looks to see if page b is contained in its own local memory. If so, then the memory access is done immediately. If not, p determines some processor q that does hold page b, and transmits a memory request to q over the network. The communication cost is dependent on the interconnection distance between p and q. Processor q services the request, and transmits back to p the (updated) value at address a.

Having a given virtual page stored at multiple processors reduces communication overhead during memory reads, but introduces the problem of maintaining consistency among the multiple copies during writes. Most multiprocessors do not provide mechanisms for maintaining consistency [5]. Therefore, various network designers have studied the *page migration* problem [4], [5], [17], which arises when each writeable page is restricted to a single copy. Suppose the single copy of page b is initially located at processor q. If the page is going to be accessed frequently by processor p, then migrating the page from p to q will reduce the communication overhead. On the other hand, moving a full page of memory incurs a large amount of communication overhead, proportional to the size of the page. In addition, moving the page to p may increase the cost of satisfying future memory requests should the pattern of accesses change. The problem, then, is to design an on-line algorithm to schedule page migrations in response to dynamically changing access patterns.

Formally, an instance of the migration problem consists of an edge-weighted undirected graph G, and a real-valued constant, D, no smaller than 1. The graph G describes the

* Received by the editors June 3, 1991; accepted for publication (in revised form) June 1, 1993. Research partially supported by NSF grant CCR-9009753. A preliminary version of this paper appeared in the Proceedings of the DIMACS Workshop on On-line Algorithms, American Mathematical Society, February 1991.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520-8285 (westbrook@cs.yale.edu).

interconnection network; each node contains a processor. We assume the nodes are numbered from 1 to n. One node of G, denoted p, contains the page. Initially the page resides at node 1. The constant D corresponds to the size of the page. A *request at node r* is a reference by processor r to some memory address on the page. A sequence of requests is generated at the nodes of G. Each new request r is satisfied immediately at cost $\delta_{p,r}$, the shortest distance in G between p and r. After satisfying a request, the page can be moved from p to a new node p' at cost $d \cdot \delta_{p,p'}$. By definition, the distance function δ satisfies the triangle inequality: $\delta_{x,y} \leq \delta_{x,z} + \delta_{z,y}$. Since G is undirected, δ is symmetric.

For any sequence of requests, there is an optimum schedule of page movements that minimizes the total cost of requests and migrations. In general, an on-line algorithm cannot compute the optimum schedule, since it lacks knowledge of the future requests. An on-line algorithm is called *c-competitive* if, for any request sequence, its cost is no more than c times the cost of the optimum off-line cost for that sequence, for some constant c. Recently much attention has been given to *competitive analysis* of on-line algorithms [6], [13], [15], [18].

Black and Sleator [5] formalized and studied the page migration problem in the context of competitive analysis. They considered two classes of networks: uniform networks, i.e., complete graphs with each edge having length 1; and trees with arbitrary edge lengths. They developed 3-competitive deterministic algorithms for these two classes for any D. In addition, they showed that for all D, no deterministic algorithm could be better than 3-competitive, even if the graph consists of only a single edge. Little is known about deterministic algorithms for a complete graph with arbitrary edge weights. This problem seems quite hard. The best previous deterministic bound for the general case was either the $2n - 1$ bound for metrical task systems [6][1] or a $2D + 2$ bound given by a simple algorithm that moves the page to the requesting node after each request.

In this paper we use randomization to beat the deterministic lower bound and to give a fast algorithm for the general case. No randomized algorithms were known prior to this work. A randomized on-line algorithm is *c-competitive* (against an oblivious adversary) if, for any request sequence, its expected cost is no more than c times the cost of the optimum off-line cost for that sequence, for some constant c. There are also definitions of randomized competitiveness against stronger adaptive adversaries which are given in §2.

For uniform networks, we give a simple algorithm, UNIFORM, that is 2.38-competitive for large values of D. This algorithm is "barely random," in the sense that it only uses a small number of random bits during an initialization phase, and from then on runs deterministically. Such a barely random algorithm has practical value since random bits can be an expensive resource.

For the general problem (which includes trees) we develop a basic algorithm from which variants can be constructed by choosing different probability distributions. We consider two such choices. The first uses random coin flips after each request and gives an algorithm that is $(1 + \phi)$-competitive for large D, where ϕ is the golden ratio, approximately 1.62. We then show that a deterministic resetting strategy gives a barely random algorithm with a competitive ratio that also tends to $1 + \phi$, although it is always slightly larger than the competitive ratio of the first algorithm. Lastly we examine the competitiveness of a very simple coin-flipping algorithm. We show that on any network and for any D it is 3-competitive against an adaptive on-line adversary. The coin flipping algorithm is memoryless, and hence has no storage or network overhead, but needs to generate a random number every request. Applying a result of Ben-David et al. [2] to the coin-flipping algorithm and the $(1 + \phi)$-competitive algorithm mentioned above, we prove the existence of a $(3 + 3\phi)$-competitive deterministic algorithm for general networks.

[1] Technically, the metrical task bound hold for a slightly different model. See below.

Page migration is one of several problems that arise in managing data in a distributed environment. Black and Sleator [5] have studied the related problem of page replication, in which one may make multiple copies of a read-only page, and Karlin et al. [11] studied snoopy caching, which is memory management given a bus-based interconnection network. Other memory management problems have been studied in references [1], [12], [15]. Migration is related to the 1-*server with excursions* problem defined by Manasse et al. [13]. Migration and 1-server with excursion are also related to the k-*server* problems [3], [7], [13]. Practical issues and applications of page migration are discussed more fully in [4], [5], and [17]. Subsequent to the work presented here, Chrobak et al. found a $2 + 1/2D$-competitive algorithm for a single edge and for the tree topology [8], and showed that this bound is tight.

2. Competitive analysis and lower bounds. Let σ be a sequence of requests. The cost of an algorithm A on σ is denoted $\mathrm{A}(\sigma)$. We denote by OPT the off-line algorithm that achieves the optimum cost on σ. Following [6], [13] we say a deterministic algorithm A is c-*competitive* if there is a constant b such that for all request sequences σ,

$$\mathrm{A}(\sigma) \le c \cdot \mathrm{OPT}(\sigma) + b.$$

For randomized algorithms the competitiveness of an algorithm is defined with respect to an *adversary*. Following [2] and [15] we consider three kinds of adversaries.

A randomized on-line algorithm, A, is c-competitive against an *oblivious* or *weak* adversary if there is a constant b such that for all graphs and for all finite request sequences σ,

$$\mathrm{E}[\mathrm{A}(\sigma)] \le c \cdot \mathrm{OPT}(\sigma) + b.$$

The expectation is over the random choices made by the on-line algorithm. This is the natural extension of deterministic competitiveness, and models a situation in which the random choices of the algorithm do not affect the choice of requests. We will typically use "c-competitive" as an abbreviation for "c-competitive against an oblivious adversary." When an adaptive adversary is intended we will state so explicitly.

An *adaptive* adversary generates the request sequence on-line, choosing requests based on the actual moves made by the on-line algorithm. That is, an adaptive adversary $\hat{\mathrm{A}}$ is a function that takes as input a sequence of $k - 1$ requests and corresponding actions by the on-line algorithm, and outputs the kth request, up to a maximum number of requests, m. (Each adversary has its own value of m.) Since the output of $\hat{\mathrm{A}}$ depends on the random choices of the on-line algorithm, a randomized on-line algorithm and an adaptive adversary together generate a probability distribution over request sequences σ.

There are two ways to charge the adversary for the request sequence. If $\hat{\mathrm{A}}$ is an *adaptive off-line* adversary, then $\hat{\mathrm{A}}(\sigma)$ is simply $\mathrm{OPT}(\sigma)$, the optimum off-line cost for σ. If $\hat{\mathrm{A}}$ is an *adaptive on-line* adversary, then $\hat{\mathrm{A}}(\sigma)$ is the cost incurred on σ by an auxiliary on-line algorithm $M_{\hat{\mathrm{A}}}$. An on-line algorithm A is c-competitive against an adaptive adversary $\hat{\mathrm{A}}$ if there exists a constant b such that

$$\mathrm{E}\left[\mathrm{A}(\sigma) - c \cdot \hat{\mathrm{A}}(\sigma)\right] \le b.$$

The adaptive adversaries model a situation in which the random choices of the algorithm may affect the future request sequence. The adaptive on-line adversary is not entirely intuitive, in that the cost measure seems somewhat unnatural, but this adversary is useful in proving theorems about deterministic algorithms.

Black and Sleator showed that in the simple case of two processors linked by a single edge of length 1, no deterministic algorithm can be better than 3-competitive for any D.

In addition, Chrobak et al. [8] give an example of a 4-processor network on which every deterministic algorithm has a competitive ratio slightly larger than 3 when $D = 1$. By a theorem of [2], these deterministic lower bounds also apply to randomized algorithms facing an adaptive off-line adversary.

The results in this paper demonstrate that against an oblivious adversary, randomized algorithms can beat the deterministic lower bound. By considering the case of a single edge, one can show that for a given page factor D no randomized algorithm can be better than $2 + \frac{1}{2D}$-competitive against an oblivious adversary [8]. We suspect, however, that the lower bound is higher for more complicated graphs.

In §5 we show a randomized algorithm that is 3-competitive against adaptive on-line adversaries for all D. One can apply a technique from [10], [16] to show that no algorithm can do better. This lower bound follows from considering the simple case of two processors linked by a single edge of length 1. The adaptive adversary's strategy is to watch the on-line algorithm and always generate requests at the processor that does not contain the on-line algorithm's page. Thus the on-line algorithm pays 1 per request. Prior to generating the first request, the adaptive on-line adversary simulates the on-line algorithm over all possible random choices, using the above strategy to generate a probability distribution over request sequences σ of length m. From the simulation, the adversary computes the expected number of requests at processors p_1 and p_2, and expected total cost of page migrations, B, paid by the on-line algorithm. If $B > m/2$, the adversary initially places its page at the processor most often requested, and never moves it again. The expected cost to the adversary is at most $D + m/2$, while the expected cost to the on-line algorithm is at least $3m/2$. If $B \leq m/2$, the adversary always moves so as to keep its page at the processor that does *not* contain the on-line algorithm's page. In this case, the adversary pays B, while the on-line algorithm pays $B + m \geq 3B$. Thus in either case, the on-line algorithm cannot guarantee to be better than 3-competitive.

The above discussion shows that in the migration problem the oblivious, adaptive on-line, and adaptive off-line adversaries are strictly differentiated from each other in their power. This is the first natural on-line problems for which this is known to be true.

3. Uniform graphs. In this section we describe and analyze a randomized algorithm, UNIFORM, designed for uniform graphs. The uniform graph is a common and important network topology [5]. In this model, $\delta_{v,v} = 0$ for all processor nodes v, $\delta_{u,v} = 1$ for all pairs $u \neq v$, and the cost of moving the page from node u to node v is $D \geq 1$.

Before describing the algorithm, we remark that when the graph is a single edge between two nodes, this problem is identical to the two-item list update problem. Reingold et al. [10], [16] gave a simple randomized algorithm for the list-update problem, which UNIFORM parallels.

Each processor node v of the graph is given an associated counter, C_v, that takes on values between 0 and $k - 1$. The optimum value of k depends on D; we will see later how to choose k. Let $\underline{C} = (C_1, C_2, \ldots, C_n)$ be a vector of the counter values at each of the n nodes of the graph. There are k^n such vectors. Let $\underline{C}_i$ be the ith such vector.

Let U^i denote a deterministic algorithm that begins with initial counter vector $\underline{C}_i$ and processes a request at node r as follows:

1. The page request is satisfied at cost at most 1.
2. The counter at r, C_r, is incremented.
3. If C_r has value k, the page is moved to node r, if it is not already there, and C_r is reset to 0.

Prior to processing any requests, the UNIFORM algorithm chooses a deterministic algorithm U^i uniformly at random, and then runs it. The random choice is made by initializing each node counter uniformly at random, using $\Theta(n \log k)$ random bits.

THEOREM 3.1. UNIFORM *is c_k-competitive against an oblivious adversary, where k is the maximum counter value and c_k is the maximum of* $1+(k+1)/2D$ *and* $1+\frac{1}{k}(2D+(k+1)/2)$.

Proof. We show that E [UNIFORM(σ)] $\leq c_k \cdot \mathrm{OPT}(\sigma)$ for any finite request sequence σ.

To analyze the expected cost of UNIFORM we imagine that the ensemble of $m = k^n$ different deterministic algorithms U^i is being run simultaneously on the request sequence. The expected cost of UNIFORM is the total cost of the ensemble divided by m. We use the following observation: At any time, each algorithm in the ensemble has a distinct counter vector. This follows by an easy induction on the number of requests. The observation implies that at any time the number of algorithms in the ensemble with the same value of C_v is exactly m/k. Equivalently, at any time during the running of algorithm UNIFORM, $\Pr[C_v = j] = 1/k$ for $0 \leq j \leq k-1$.

Our proof uses the technique of comparing simultaneous runs of U^i and OPT on the request sequence σ. The actions of U^i and OPT are partitioned into two kinds of events, which together account for all costs to the algorithms. The first kind of event is the servicing of a request by both U^i and OPT. This event may involve U^i moving its page. The second kind of event is OPT moving its page. After the tth event, let p^i denote the node where U^i has the page and opt denote the node where OPT has the page. We define a *potential function* Φ:

$$\Phi_t = \begin{cases} 0 & \text{if } p^i = \text{opt}, \\ D+k-C_{\text{opt}} & \text{if } p^i \neq \text{opt}. \end{cases}$$

Each event has an actual cost to OPT, and an *amortized cost* to U^i, defined to be the actual cost of the event to U^i plus the change in the potential function, $\Delta\Phi = \Phi_t - \Phi_{t-1}$. The potential Φ is always nonnegative, and $\Phi_0 = 0$, since OPT and U^i begin with the page at the same node. This implies, by standard amortized analysis arguments, that the total actual cost to U^i is bounded by the sum of the amortized costs for all events. To prove the theorem, we show that for each event the sum of the amortized costs to the algorithms in the ensemble is less than mc_k times the actual cost to OPT.

1. The event is a request. There are two subcases.

(i) The request is at node opt. In this case OPT pays 0, so we must show that the amortized cost to each algorithm U^i is also zero. If $p^i =$ opt then the actual and amortized cost to U^i is also 0. Otherwise, U^i pays 1, but since C_{opt} is incremented, the potential decreases by 1. If the counter reaches k, then the page moves to opt. This costs D, but decreases the potential by a further D. Thus the amortized cost is always 0.

(ii) The request is at node $r \neq$ opt. The cost to OPT is 1. Each algorithm U^i pays at most 1 for the request. For m/k of the algorithms in the ensemble, C_r reaches k and the page is moved to r at cost D. If $p^i \neq$ opt prior to moving, then there is no change in the potential. In the worst case, $p^i =$ opt prior to the move and the potential increases by $D+k-C_{\text{opt}}$. No more than m/k^2 algorithms, however, can move from opt with $C_{\text{opt}} = i$, for $0 \leq i \leq k-1$. Therefore, the total cost of the event to the ensemble is bounded by

$$m + D\frac{m}{k} + \sum_{0 \leq i \leq k-1} \frac{m}{k^2}(D+k-i).$$

Summing and dividing by m gives the expected amortized cost to UNIFORM,

$$1 + \frac{1}{k}\left(2D + \frac{k+1}{2}\right),$$

which is bounded by c_k times the cost to OPT.

2. The event is that OPT moves its page to a new node r. The actual cost to OPT is D, and the actual cost to each algorithm U^i is 0. Now we consider the cost due to changes in potential. If $r = p^i$, the potential decreases by at least D. If $r \neq p^i$ and opt $\neq p^i$, then $\Delta\Phi = C_{\text{opt}} - C_r$, which is bounded by $k - C_r$. In the worst case, $r \neq p^i$ and initially opt $= p^i$. Then $\Delta\Phi = D + k - C_r$. Exactly m/k algorithms have $C_r = i$, so the total cost to the ensemble of this event is at most

$$\sum_{0 \le i \le k-1} \frac{m}{k}(D + k - i).$$

Summing and dividing by m gives the expected cost to UNIFORM, $D + (k+1)/2$, which is at most c_k times the cost to OPT. □

This completes the proof of Theorem 3.1. Given a value of D, we can choose k to minimize the maximum of $1 + (k+1)/2D$ and $1 + \frac{1}{k}(2D + (k+1)/2)$.

Table 1 shows the best competitive ratio for UNIFORM for values of D up to 10. These values are found by setting

$$1 + \frac{k+1}{2D} = 1 + \frac{1}{k}\left(2D + \frac{k+1}{2}\right)$$

and solving for k in terms of D. Then the best integer approximation to this value is taken. As D tends to infinity, the best competitive ratio decreases and tends to $(5 + \sqrt{17})/4 \approx 2.28$. Note that all these values are better than the deterministic lower bound of 3. It is possible to slightly improve upon the values for small D by using the random reset techniques employed in [16].

TABLE 1
Best competitive ratios for UNIFORM.

D	best k	comp. ratio
1	2	2.75
2	5	2.50
3	7	2.43
4	10	2.38
5	12	2.38
6	15	7/3
7	17	2.35
8	20	2.33
9	23	2.33
10	25	2.32

The next theorem shows that our analysis is tight.

THEOREM 3.2. *The UNIFORM algorithm is no better than c_k-competitive, where c_k is defined as in Theorem* 3.1.

Proof. Consider two processors, labelled "1" and "2," connected by a single edge of length 1. Initially the page is located at processor 1. There are two kinds of bad request sequences: $\sigma_a = 2^k$ and $\sigma_b = 21^k$, where 1 and 2 denote page requests at processors 1 and 2 respectively, and i^k denotes k consecutive requests at processor i. Sequence σ_a can be satisfied at cost D, and σ_b can be satisfied at cost 1.

The expected cost to UNIFORM on σ_a is simply the expected time until the page migrates to 2, which is determined by the expected value of the counter at 2, plus the cost of migration. (UNIFORM is guaranteed to move the page eventually, since there are k requests.) Therefore

$$\mathrm{E}\left[\mathrm{UNIFORM}(\sigma_a)\right] = D + \frac{k+1}{2},$$

or $(1 + (k + 1)/2D) \cdot \mathrm{OPT}(\sigma_a)$.

On σ_b, UNIFORM moves its page to 2 after the first request with probability $1/k$, in which case the remainder of the sequence looks identical to σ_a. Hence

$$\mathrm{E}\left[\mathrm{UNIFORM}(\sigma_a)\right] = \left(1 + \frac{1}{k}(2D + \frac{k+1}{2})\right) \cdot \mathrm{OPT}(\sigma_b).$$

The adversary can generate arbitrarily long request sequences by repeating σ_a or σ_b, so the UNIFORM algorithm cannot be better than c_k-competitive. □

4. General graphs. In this section we consider the migration problem on undirected graphs with arbitrary edge distances. Recall that even with arbitrary edge lengths the shortest distance function δ remains symmetric and still satisfies the triangle inequality.

One reason the general case seems harder than the uniform case is that a general graph may contain clusters of processors that are very close to each other but relatively far from other clusters. Such a situation models a collection of local-area networks tied together in a wide-area network. Even though no single processor in a cluster is accessing the page especially often, the cluster as a whole may generate many requests for the page, and it may be advantageous to move the page to some node in the cluster. On the other hand, in the uniform case a cluster of processors that access the page equally often is an easy situation for the on-line algorithm to handle, since both it and OPT must pay about 1 per request.

The previous best known deterministic competitive ratio for general graphs is $2n - 1$, given by the general metrical task system algorithm [6]. Technically, that result holds for a different model, the "lookahead-1" model, in which the algorithm is allowed to move the page after seeing the next request but before actually servicing it. In our "lookahead-0" model, the request must be serviced immediately it is seen. It is easily checked that an algorithm for the lookahead-1 model can be used to derive an algorithm for the lookahead-0 model that has a competitive ratio no more than 2 times worse than that of the original algorithm. Furthermore, the bounds converge as D grows large. One can also show that the strategy of moving the page to the last requesting node yields a competitive ratio of $2D + 2$. Thus even when the graph is a single triangle, the best known deterministic algorithm for large D is only 5-competitive. Our randomized algorithms for the general case are simple enough to be practical, and beat the deterministic lower bound.

The general algorithm G maintains a *single* counter with value $C \geq 1$. After each request to the page, we perform the following.

1. The counter is decremented by one, regardless of where the request is.

2. If the counter reaches 0, the page is moved to the location of the current request. After the move, the counter is reset according to a *resetting distribution* $\mathcal{D} = (\alpha_1, \alpha_2, \alpha_3, \ldots)$, where α_i is the probability of resetting the counter to value i.

The distribution $\mathcal{D}$ is fixed in advance and is independent of the request sequence. Let $k = \max\{i \mid \alpha_i > 0\}$. We are only interested in distributions for which k exists (i.e., is not infinite).

The counter value forms a Markov chain in which state i corresponds to the counter having value i, for $1 \leq i \leq k$. Being in state i means that the page will be moved after i accesses. A state transition occurs with every access. The transition matrix for the corresponding Markov chain is shown in Fig. 1. Note that the probability of being in a given state depends only on the number of requests that have been processed, not on the nature of the requests.

Let ρ_i denote the steady state probability of being in state i. Given a resetting distribution $\mathcal{D}$, let $\bar{C}_R(\mathcal{D})$ denote the expected value of the counter immediately following a reset and let $\bar{C}_S(\mathcal{D})$ denote the expected value of the counter in steady state. By definition,

$$\begin{pmatrix} \alpha_1 & 1 & 0 & 0 & \dots & 0 \\ \alpha_2 & 0 & 1 & 0 & & 0 \\ \alpha_3 & 0 & 0 & \ddots & & 0 \\ \vdots & & & & & \\ \alpha_{k-1} & & & & & 1 \\ \alpha_k & & & & & 0 \end{pmatrix}$$

FIG. 1. *Markov chain transition matrix. Entry* (i, j) *contains the probability of a transition to counter value* i *from counter value* j *following a request,* $1 \le i, j \le k$.

$\bar{C}_R(\mathcal{D}) = \sum_{i=1}^{k} i\alpha_i$ and $\bar{C}_S(\mathcal{D}) = \sum_{i=1}^{k} i\rho_i$. We will abbreviate the notation to $\bar{C}_R$ and $\bar{C}_S$ when the distribution $\mathcal{D}$ is clear.

The steady state probabilities ρ_i are given by the eigenvector of the transition matrix corresponding to eigenvalue 1. One may verify that in steady state,

$$\rho_i = \frac{1}{\bar{C}_R} \sum_{j=i}^{k} \alpha_j .$$

This implies

$$\begin{aligned} \bar{C}_S &= \frac{1}{\bar{C}_R} \left(\sum_{i=1}^{k} i \sum_{j=1}^{k} \alpha_j \right) \\ &= \frac{1}{\bar{C}_R} \left(\sum_{i=1}^{k} \alpha_i \sum_{j=1}^{i} j \right) \\ &= \frac{1}{\bar{C}_R} \left(\sum_{i=1}^{k} i(i+1)\alpha_i \right). \end{aligned}$$

Prior to starting a run of the algorithm, the counter is initialized according to the steady state distribution, i.e., the counter is initialized to i with probability ρ_i, for $1 \le i \le k$.

LEMMA 4.1. *Let* $\mathrm{G}(\mathcal{D})$ *be the general algorithm with resetting distribution* $\mathcal{D}$. $\mathrm{G}(\mathcal{D})$ *is* $c_{\mathcal{D}}$*-competitive, where* $c_{\mathcal{D}}$ *is the maximum of* $2 + (2D/\bar{C}_R(\mathcal{D}))$ *and* $1 + (\bar{C}_S(\mathcal{D})/D)$.

Proof. As before, we partition the actions and costs into two events: a request that both $\mathrm{G}(\mathcal{D})$ and OPT satisfy, and a page movement by OPT. A given σ fixes a sequence of events and determines an ensemble of deterministic on-line algorithms that correspond to all possible random choices that can be made throughout the processing of σ. At any time, the fraction of algorithms in this ensemble that have counter value i is given by the steady state probability, ρ_i. The average cost of the algorithms in this ensemble gives the expected cost of one run of G on σ. We will calculate the expected amortized cost per operation of G directly. Let random variable X_t be the amortized cost of the tth event. The total amortized cost is $\sum_t X_t$, and by linearity of expectations the expected total amortized cost is $\sum_t \mathrm{E}[X_t]$. To prove the theorem we bound $\mathrm{E}[X_t]$ in terms of the cost to OPT.

At the time of event t, let opt be the node where OPT has its page and g be the node where G has its page. Let C be the value of the counter. Both g and C depend upon previous random choices, but opt is fixed by σ.

We use the potential function

$$\Phi_t = (D + C) \cdot \delta_{g,\mathrm{opt}}.$$

Note that $\Phi = 0$ when OPT and G have their pages at the same node.

Now we analyze the expected amortized cost of each event.

1. The event is a request at node r. Let $\ell_0 = \delta_{r,\mathrm{opt}}$, $\ell_1 = \delta_{g,\mathrm{opt}}$ and $\ell_2 = \delta_{g,r}$. (See Fig. 2.) The cost to OPT is ℓ_0. The actual cost to G is ℓ_2. The counter is decremented, so $\Delta\Phi = -\ell_1$, and hence the amortized cost to G is $\ell_2 - \ell_1$. By the triangle inequality, $\ell_2 \leq \ell_0 + \ell_1$. Therefore, the amortized cost to G to satisfy the request is at most ℓ_0.

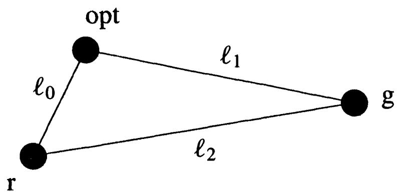


FIG. 2. *Diagram for proof of Lemma* 4.1.

With probability ρ_1, the counter value is 1 just prior to the request and becomes 0 after the request is satisfied. Then G moves the page to node r at cost $D\ell_2$ and resets the counter according to $\mathcal{D}$. The potential change is

$$\ell_0(D + C') - D\ell_1,$$

where C' is the value to which the counter is reset. The *expected* change in potential is determined by the expected value of the counter after the reset, $\bar{C}_R$. Hence the expected amortized cost of a move is

$$D\ell_2 + \ell_0(D + \bar{C}_R) - D\ell_1.$$

Again applying the triangle inequality, this is bounded by

$$\ell_0(2D + \bar{C}_R).$$

Combining the cost to satisfy the request with the expected move cost in the case that $C = 1$, the total expected amortized cost to G is bounded by

$$\ell_0 \left(1 + \rho_1(2D + \bar{C}_R)\right).$$

Since $\rho_1 = (1/\bar{C}_R)$, this is at most $c_{\mathcal{D}}$ times the cost to OPT.

2. The event is OPT moving its page from node opt to a new node opt$'$. Let $\ell_0 = \delta_{\mathrm{opt},\mathrm{opt}'}$. The actual cost to OPT is $D\ell_0$ and the actual cost to G is 0. Now we consider the cost due to changes in potential. Let ℓ_1 be $\delta_{g,\mathrm{opt}}$ and ℓ_2 be $\delta_{g,\mathrm{opt}'}$. The actual potential change is

$$(\ell_2 - \ell_1) \cdot (D + C),$$

where C is the actual counter value. The *expected* potential change is determined by the steady state expected value of the counter, $\bar{C}_S$. Since $\ell_2 \leq \ell_0 + \ell_1$, the expected potential change is bounded by $\ell_0 \cdot (D + \bar{C}_S)$, which is at most $c_{\mathcal{D}}$ times the cost to OPT. □

LEMMA 4.2. *The general algorithm,* $G(\mathcal{D})$*, is no better than* $c_{\mathcal{D}}$*-competitive, where* $c_{\mathcal{D}}$ *is defined as in Lemma* 4.1.

Proof. As in Theorem 3.2 we consider two processors, connected by an edge of length 1, and two request sequences: $\sigma_a = 2^k$ and $\sigma_b = 21^k$.

The expected cost to $G(\mathcal{D})$ on σ_a is $\bar{C}_S$, the initial expected time until the page migrates to 2, plus the cost of the migration. Therefore

$$E[G(\sigma_a)] = D + \bar{C}_S,$$

or $(1 + (\bar{C}_S)/D) \cdot \mathrm{OPT}(\sigma_a)$.

On σ_b, $G(\mathcal{D})$ moves its page to 2 after the first request with probability ρ_1. In this case the cost on the remainder of σ_b is determined by the expected time to return the page to 1, given that it has moved initially. This expected time is $\bar{C}_R$. Hence

$$E[G(\sigma_a)] = (1 + \rho_1(2D + \bar{C}_R)) \cdot \mathrm{OPT}(\sigma_b) = \left(2 + \frac{2D}{\bar{C}_R}\right)\mathrm{OPT}(\sigma_b).$$

The adversary can generate arbitrarily long request sequences by repeating σ_a or σ_b. □

In the next two subsections we analyze two choices for the resetting distribution $\mathcal{D}$. First we give a randomized resetting strategy with a competitive ratio that tends to $1 + \phi$, where ϕ is the golden ratio, approximately 1.618. Then we discuss the deterministic strategy of resetting the counter to a single value k after a move. We show that the competitive ratio of the deterministic strategy approaches that of the random resetting distribution as D grows large. Deterministic resetting gives a simple, practical, near-optimal barely random algorithm.

4.1. Random resetting. We restrict our attention to distributions $\mathcal{D}$ that satisfy

$$2 + \frac{2D}{\bar{C}_R} = 1 + \frac{\bar{C}_S}{D}. \tag{1}$$

Then the competitive ratio is given by the choice of resetting probabilities that minimizes $2 + 2D/\bar{C}_R$. This is minimized when $\bar{C}_R$ is maximized. For a fixed value of k, the following linear program computes an optimum resetting distribution subject to constraint 1:

$$\text{maximize } \sum_{i=1}^{k} i\alpha_i,$$

subject to the constraints

$$\sum_{i=1}^{k} \alpha_i = 1,$$

$$\sum_{i=1}^{k} (i^2 - (2D-1)i)\alpha_i = 4D^2,$$

$$\alpha_i \geq 0, \qquad 1 \leq i \leq k.$$

Let x be the positive solution to $x^2 - (2D-1)x = 4D^2$. Hence $x = D + \frac{1}{2}((20D^2 - 4D + 1)^{1/2} - 1)$. Then $k_0 = \lceil x \rceil$ is the least integer such that constant on the k_0th term in the second constraint of the linear program is greater than $4D^2$.

LEMMA 4.3. *There is a feasible solution to the linear program with only* α_{k_0-1} *and* α_{k_0} *nonzero.*

Proof. Let $\alpha = \alpha_{k_0-1}$. By the first constraint of the linear program, $\alpha = 1 - \alpha_{k_0}$. Setting $\alpha_i = 0$ for all other i, the second constraint becomes

$$0 = k_0^2 - (2D - 1)k_0 - 4D^2 + \alpha(2D - 2k_0).$$

Replacing k_0 by $x + \epsilon$, where $0 \leq \epsilon < 1$, and using the fact that $x^2 - (2D - 1)x = 4D^2$, this constraint further reduces to

$$0 = (2x - 2D + 1 + \epsilon)\epsilon - (2x - 2D + 2\epsilon)\alpha,$$

implying

$$\alpha = \epsilon + \epsilon(1 - \epsilon)/(2x - 2D + 2\epsilon).$$

Observing that $2x - 2D \geq 0$ for all $D \geq 0$, it is easily verified that $0 \leq \alpha \leq 1$. Hence constraint three is satisfied. □

THEOREM 4.4. *The competitive ratio of the algorithm given by choosing k_0 and α as above is* $2 + 2D/(k_0 - \alpha)$

Proof. This theorem follows from straightforward substition into the formula for $\bar{C}_R$. □

Table 2 shows several competitive ratios given by Theorem 4.4. As D gets large, the best competitive ratio decreases and tends to $1 + (1 + \sqrt{5})/2$, which is one plus the golden ratio, or approximately 2.62.

TABLE 2
Best general competitive ratios, resetting randomly with α_{k_0-1} and α_{k_0}.

D	best k_0	comp. ratio
1	3	2.800
2	6	2.696
3	9	2.667
4	13	2.655
8	26	2.636
16	52	2.627
32	103	2.622
64	207	2.620
128	414	2.619
256	828	2.619
512	1657	2.618

4.2. Deterministic resetting. We can turn the randomized resetting strategy of the previous section into a deterministic resetting strategy by setting either α_{k_0-1} or α_{k_0} to 1, depending on which gives the lowest maximum competitive ratio. Let $s = k_0 - 1$ or k_0 depending on whether we choose $\alpha_{k_0-1} = 1$ or $\alpha_{k_0} = 1$, respectively.

THEOREM 4.5. *The general algorithm with deterministic resetting is c_s-competitive against an oblivious adversary, where c_s is the maximum of* $1 + (s+1)/2D$ *and* $2 + (2D/s)$.

Proof. The counter is initialized by setting it to value i with probability $1/s$ for $1 \leq i \leq s$. At any time thereafter, the counter values are uniformly distributed from 1 to s. Calculating $\bar{C}_R$ and $\bar{C}_S$ from the formulas given above and applying Lemma 4.1 completes the proof. □

Table 3 shows the best competitive ratios achieved by this deterministic resetting strategy for various values of D. As D and hence k_0 grows, the deterministic and random resetting algorithms tend to the same competitiveness. Deterministic resetting has the advantage of greater simplicity, since it requires random bits only during initialization.

TABLE 3
Best general competitive ratios, resetting to k_0 with probability 1.

D	best k_0	comp. ratio
1	3	3.000
2	6	2.750
3	9	2.667
4	12	2.667
8	25	2.640
16	51	2.627
32	103	2.625
64	206	2.621
128	413	2.620
256	827	2.619
512	1656	2.618

5. The coin-flipping algorithm. A very simple randomized strategy is to flip a weighted coin at each request, and move to the requested node if the coin comes up heads. That is, at each request, the page is moved to the requesting node with probability $p > 0$. This strategy can be used on a general graph with arbitrary edge weights. It has the advantage of being *memoryless*; no state information is remembered other than that given by the location of the page. Furthermore, it achieves exactly a competitive ratio of 3 against both adaptive on-line adversaries and oblivious adversaries. On the other hand, it has a poorer competitiveness against an oblivious adversary than the algorithms for the uniform and general problems presented in §§3 and 4, respectively, and it requires a random number call every access.

In this section we analyze the competitiveness of the coin-flipping algorithm. First we give a lower bound.

THEOREM 5.1. *The coin-flipping algorithm is no better than* 3*-competitive against an oblivious adversary.*

Proof. We use our standard scenario of two processors and two request sequences: $\sigma_a = 2^j$ and $\sigma_b = 21^j$. We require that $j > D$. For σ_a the optimum cost is D, and for σ_b the optimum cost is 1.

Let $q = 1-p$. The expected cost of the coin-flipping algorithm on σ_a is $(D+1/p)(1-q^j)$. For any $p \leq 1/2D$, this is at least $3(1-q^j)$ times the optimal cost. The expected cost of the coin-flipping algorithm on σ_b is $1 + p(D+(D+1/p)(1-q^j)) \geq 2pD+2(1-q^j)$. For any $p \geq 1/2D$, this is at least $3-2q^j$ times the optimal cost. Hence for any choice of p, there is a sequence and a choice of j such that the competitive ratio of the coin-flipping algorithm is arbitrarily close to 3. □

Now we show that the coin-flipping algorithm is 3-competitive against an adaptive on-line adversary.

To perform the analysis, we use the request-answer game framework of [2]. The adversary initially commits to a request sequence length m. The game proceeds as a sequence of m rounds. In each round, the adversary first moves its page, if it so chooses. Then the adversary generates a request, and immediately services it. Then the on-line algorithm services the request and possibly moves its page. At the end of each round, the adversary is told the location of the page of the on-line algorithm.

THEOREM 5.2. *The coin-flipping algorithm with $p = 1/2D$ is* 3*-competitive against any adaptive on-line adversary $\hat{A}$.*

Proof. We use the following crucial observation. In each round, the probability that the coin-flipping algorithm moves its page is independent both of the request and of the location of the adversary's page when the request is generated. This observation is not true for the algorithms of §§3 and 4, which make moves based on counter values that the adaptive adversary knows and can use in generating its requests. This observation is also not true for the coin-flipping algorithm against an adaptive off-line adversary. In the off-line case, the location of the adversary's page depends deterministically upon future requests, which in turn depend upon the outcomes of future coin-flips by the algorithm.

Using the observation, we show that the expected amortized cost of each event is no more than 3 times the cost to the adversary of that event. Let f denote the processor containing the page of the coin flipping algorithm, a denote the processor containing the page of $\hat{A}$, and ℓ denote the distance from f to a. We use the following potential function:

$$\Phi = 3D\ell.$$

There are two events to consider in each round:

1. A request occurs at node r. Let f and a denote the nodes containing the pages of the coin-flipping algorithm and the adversary, respectively, just prior to the request. There are three distances of interest: ℓ_0, the distance from f to a; ℓ_1, the distance from a to r; and ℓ_2, the distance from f to r. The cost to $\hat{A}$ is ℓ_1. The expected cost to the coin-flipping algorithm is

$$\ell_2 + \frac{1}{2D}(D\ell_2 + 3D\ell_1 - 3D\ell_0).$$

By the triangle inequality, $\ell_2 \leq \ell_1 + \ell_0$. This implies that the expected cost to the coin flipping algorithm is at most $3\ell_1$.

2. $\hat{A}$ moves its page from a to a'. Let ℓ_0 and ℓ_1 denote the distance from f to a and f to a', respectively. Let ℓ_2 be the distance from a to a'. The cost to $\hat{A}$ of the move is $D\ell_2$. The cost to the coin flipping algorithm is $3D\ell_1 - 3D\ell_0$, which is at most $3D\ell_2$, by the triangle inequality. □

COROLLARY 5.3. *The coin-flipping algorithm is* 3*-competitive against an oblivious adversary.*

Proof. This follows trivially from the fact that an oblivious adversary is simply an adaptive on-line adversary that generates the same requests regardless of the random choices made by the on-line algorithm and services these requests optimally. □

COROLLARY 5.4. *On any graph the competitiveness of the coin-flipping algorithm against an adaptive off-line adversary tends to* $3 + 3\phi \approx 7.86$ *as D grows large.*

Proof. Theorem 2.2 of [2] states that if algorithm B is β-competitive against an adaptive on-line algorithm and there exists an algorithm A that is α-competitive against an oblivious adversary, then B is $\alpha\beta$-competitive against an adaptive off-line algorithm. We take A to be the general algorithm with deterministic resetting from §4.2 and B to be the coin-flipping algorithm. For various values of D, the competitive factor of the coin-flipping algorithm against an adaptive off-line adversary can be found by multiplying by 3 the appropriate entries in Table 3. □

COROLLARY 5.5. *There exists a deterministic algorithm for the migration problem with competitive ratio that tends to* $3 + 3\phi \approx 7.86$ *as D grows large.*

Proof. Theorem 2.1 of [2] states that if there is a randomized algorithm that is α-competitive against any adaptive off-line algorithm, then there is also a α-competitive deterministic algorithm. The theorem gives a method of constructing the deterministic algorithm, but we have not attempted to do so here. □

6. Remarks. This paper presented randomized algorithms for page migration. We have shown the following.

(i) An algorithm for uniform graphs that tends toward being 2.38-competitive as D gets large.

(ii) An algorithm for general graphs that is parameterized by a resetting distribution. Deterministic resetting gives a variant that tends toward being $1 + \phi$-competitive as D gets large.

(iii) A very simple coin-flipping algorithm that is 3-competitive against adaptive on-line adversaries.

We have also shown that the analysis is tight for each algorithm and that coin-flipping is optimal against an adaptive on-line adversary.

There are several interesting open problems. One is to improve upon our randomized algorithms, and to improve or generalize the lower bounds. A second one is to find a good deterministic algorithm for migration on general graphs. Another open problem is to extend any of these results to the more general problem of 1-server with excursions.

In recent work, Bartal et al. [1] studied file allocation in distributed systems. In this situation multiple physical copies of writeable pages or data files are sometimes allowed. In this problem, the competitive ratio rises to $\Theta(\log n)$, where n is the number of processors.

Acknowledgments. The author thanks Daniel D. Sleator for suggesting the migration problem and Nick Reingold for helpful conversations.

REFERENCES

[1] Y. Bartal, A. Fiat, and Y. Rabani, *Competitive algorithms for distributed data management*, in Proc. ACM Symp. on Theory of Computing, 1992, pp. 39–50.

[2] S. Ben-David, A. Borodin, R. M. Karp, G. Tárdos, and A. Wigderson, *On the power of randomization in on-line algorithms*, in Proc. 22nd ACM Symp. on Theory of Computing, ACM Press, New York, NY, May 1990, pp. 379–386.

[3] P. Berman, H. J. Karloff, and G. Tardos, *A competitive three-server algorithm*, in Proc. 1st ACM-SIAM Symp. on Discrete Algorithms, 1990, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 280–290.

[4] D. Black, A. Gupta, and W. Weber, *Competitive management of distributed shared memory*, in Proc. Spring Compcon 1989, IEEE Computer Society Press, San Francisco, CA, 1989, pp. 184–190.

[5] D. L. Black and D. D. Sleator, *Competitive algorithms for replication and migration problems*, Tech. Report CMU-CS-89-201, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1989.

[6] A. Borodin, N. Linial, and M. Saks, *An optimal online algorithm for metrical task systems*, in Proc. 19th ACM Symp. on Theory of Computing, ACM Press, New York, NY, 1987, pp. 373–382.

[7] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, *New results on server problems*, in Proc. 1st ACM-SIAM Symp. on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990, pp. 291–300.

[8] M. Chrobak, L. L. Larmore, N. Reingold, and J. Westbrook, *Page migration algorithms using work functions*, Tech. report YALEU/DCS/TR-897, Yale University, New Haven, CT, November 1991.

[9] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, *Performance measurements on a* 128-*node butterfly parallel processor*, in Proc. International Conf. on Parallel Processing, IEEE Computer Society Press, Silver Spring, MD, 1985, pp. 531–540.

[10] S. Irani, N. Reingold, D. D. Sleator, and J. Westbrook, *Randomized algorithms for the list update problem*, in Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991, pp. 251–260.

[11] A. Karlin, M. Manasse, L. Rudolph, and D. Sleator, *Competitive snoopy caching*, Algorithmica, 3 (1988), pp. 79–119.

[12] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki, *Competitive randomized algorithms for non-uniform problems*, in Proc. 1st ACM-SIAM Symp. on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1990, pp. 301–309.

[13] M. Manasse, L. A. McGeoch, and D. Sleator, *Competitive algorithms for on-line problems*, in Proc. 20th ACM Symp. on Theory of Computing, ACM Press, Chicago, IL, 1988, pp. 322–333.

[14] G. PFISTER AND ET AL., *The IBM research parallel processor prototype: Introduction and architecture*, in Proc. International Conf. on Parallel Processing, IEEE Computer Society Press, Silver Spring, MD, 1985, pp. 764–771.

[15] P. RAGHAVAN AND M. SNIR, *Memory versus randomization in on-line algorithms*, Research Report RC 15622, No. 69444, IBM T. J. Watson Reseach Center, 1990.

[16] N. REINGOLD, J. WESTBROOK, AND D. D. SLEATOR, *Randomized algorithms for the list update problem*, Algorithmica, 11 (1994), pp. 15–32.

[17] C. SCHEURICH AND M. DUBOIS, *Dynamic page migration in multiprocessors with distributed global memory*, IEEE Trans. Comput., 38 (1989), pp. 1154–1163.

[18] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.

[19] A. WILSON, *Hierarchical cache/bus architecture for shared memory multiprocessors*, in Proc. 14th International Symp. on Computer Architecture, ACM SIGARCH/IEEE Computer Society, IEEE Computer Society Press, New York, 1987, pp. 244–252.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 966–975, October 1994

NEAR-OPTIMAL TIME-SPACE TRADEOFF FOR ELEMENT DISTINCTNESS*

ANDREW CHI-CHIH YAO†

Abstract. It was conjectured in Borodin et al. [J. Comput. System Sci., 22 (1981), pp. 351–364] that to solve the element distinctness problem requires $TS = \Omega(n^2)$ on a comparison-based branching program using space S and time T, which, if true, would be close to optimal since $TS = O((n \log n)^2)$ is achievable. Recently, Borodin et al. [SIAM J. Comput., 16 (1987), pp. 97–99] showed that $TS = \Omega(n^{3/2}(\log n)^{1/2})$. This paper presents a near-optimal tradeoff $TS = \Omega(n^{2-\epsilon(n)})$, where $\epsilon(n) = O(1/(\log n)^{1/2})$.

Key words. branching programs, comparisons, element distinctness, linear ordering, lower bounds, time-space tradeoff

AMS subject classifications. 68P10, 68Q25

1. Introduction. In Cobham's classic paper [C], time-space tradeoffs were established for one-tape Turing machines. In recent years, a number of time-space tradeoff results have been obtained for various computational models, such as Boolean and arithmetic circuits [To], a general sequential computing model [BC], [B], multihead Turing machines [DG], [K], comparison-based branching programs [BFKLT], [Y], [BFMUW], [K], and very large scale integration (VLSI) models [Th] (see [U] for a review of results). In this paper, we establish a tradeoff result in the comparison-based branching program model, proving in weaker form an interesting conjecture of Borodin et al. [BFKLT].

Borodin et al. [BFKLT] proved a tradeoff $TS = \Omega(n^2)$ for sorting n numbers on a comparison-based branching program, but were not able to establish a similar tradeoff for any decision problem in their model. They conjecture, however, that the tradeoff $TS = \Omega(n^2)$ is also true for the element distinctness problem. This, if proved, would be close to the best possible, since an upper bound $TS = O((n \log n)^2)$ is achievable for sorting [MP], and hence for the element distinctness problem. Recently, Borodin et al. [BFMUW] gave a partial resolution to the above conjecture, showing in the same model that $TS = \Omega(n^{3/2}(\log n)^{1/2})$. In this paper, we will prove that $TS = \Omega(n^{2-\epsilon(n)})$, where $\epsilon(n) = O(1/(\log n)^{1/2})$. As mentioned earlier, such a tradeoff is nearly the best possible.

Let $x_1, x_2, \dots, x_n$ be n elements chosen from a linearly ordered set $(D, \leq)$. The element distinctness problem (with parameter n) is to decide whether all x_i are distinct. As discussed in [BFMUW], a *comparison branching program A* is a labeled directed acyclic graph with a distinguished nonsink node, called *source*. Each nonsink node is labeled by a comparison $x_i : x_j$ with $i \neq j$, and has three outgoing edges labeled by $<, =, >$, respectively. The sinks are labeled by either "accept" or "reject." An input $\tilde{x} = (x_i, x_2, \dots, x_n) \in D^n$ starts at the source and traverses A, making comparisons and branching according to the outcomes, until a sink is reached. The input is accepted if and only if it reaches a sink with an "accept" label. The *capacity* of A is the base-2 logarithm of the number of nodes. The *length* of A, or the *time T* used by A, is the length of the longest path starting with source. We say that A is an *algorithm* for *the element distinctness problem* if $\tilde{x}$ is accepted when and only when all x_i are distinct. Let $\mathcal{A}_n$ denote the set of all algorithms for the element distinctness problem. We now state our main result.

THEOREM 1. *Any* $A \in A_n$ *with capacity S and time T must satisfy* $TS = \Omega(n^{2-\epsilon(n)})$ *for large n, where* $\epsilon(n) = 5/(\ln n)^{1/2}$.

*Received by the editors March 29, 1988; accepted for publication (in revised form) June 1, 1993. This research was supported in part by National Science Foundation grant CCR-8813283.

†Department of Computer Science, Princeton University, Princeton, New Jersey 08544 (yao@cs.princeton, edu).

2. Preliminaries.

2.1. Overview. As discussed in [BFKLT], using a result of Nick Pippenger, we can assume without loss of generality that A is *leveled*, i.e., each node is assigned a level number and each edge goes from a node in level i to a node in level $i+1$. The source node has level 0, and all sinks are at level T. From now on, all branching programs will mean leveled comparison branching programs.

We review the ideas involved in the proof of $TS = \Omega(n^{3/2}(\log n)^{1/2})$ in [BFMUW]. (Some of these ideas originated in [BFKLT].) Let $A \in \mathcal{A}_n$. For any input $\tilde{x} = (x_1, x_2, \ldots, x_n)$ with distinct x_i, the sequence of comparisons made by A must include all the "adjacent" ones, i.e., comparisons of the form $x_{i_j} : x_{i_{j+1}}$ if the input satisfies $x_{i_1} < x_{i_2} < \cdots < x_{i_n}$; otherwise we could have two identical x_r. The idea is to show that any branching program of length less than or equal to n_0, where $n_0 = (nS/(16e))^{1/2}$, can make more than S adjacent comparisons only for a very small fraction of the $n!$ possible linear orderings in the input. Thus, A must have $(n-1)/S$ consecutive blocks of levels, each block of length n_0, to perform the needed $n-1$ adjacent comparisons for all linear orderings.

To prove Theorem 1, we will adopt the same general approach. We will show that any branching program of length less than or equal to $n_1 = n^{1-\epsilon(n)}$, can make more than $O(S \cdot n^{\epsilon(n)})$ adjacent comparisons only for a very small fraction of the possible linear orderings. The asserted tradeoff then follows the same line of reasoning as before.

We first give some intuitive ideas behind the proof. In [BFMUW], the main technical lemma can be roughly described as follows. Given T inequalities of the form $x_i < x_j$, if we take a random linear ordering of the x_k's, the expected number of these inequalities being adjacent comparisons is $O(T^2/n)$. The bound $O(T^2/n)$ is tight in some cases, such as when the sequence of inequalities are $x_1 < x_2 < \cdots < x_{T+1}$. However, in many other cases, this bound is too large an overestimate. For example, if the inequalities are $x_1 < x_2, x_3 < x_4, \ldots, x_{2T-1} < x_{2T}$, then the expected number of these inequalities being adjacent comparisons is $O(T/n)$. We have an opportunity to derive an improved bound, if a branching program of length T can only infrequently produce T inequalities with the $O(T^2/n)$ behavior.

To see how such a plan might work, consider any branching program of length $o(n^{5/3})$ and $O(\log n)$ capacity. For simplicity, we will ignore logarithm factors in the rest of this overview. Divide the program into blocks of length $n^{2/3}$. Assume that associated with each block b there is a subset D_b of $O(n^{2/3})$ variables x_i such that, within the block b, all comparisons made are between variables in D_b. We will argue informally that this branching program cannot solve the element distinctness problem. For each block b, we can regard b as a branching program working on the input numbers in D_b. Using the original argument in [BFMUW], one can easily show that typically the sequence of inequalities produced by the block b will have an expected $O(n^{1/3})$ comparisons that are *adjacent within* D_b (for a random linear ordering among the elements in D_b). Now, when we throw in the $n - O(n^{2/3}) x_i$ outside of D_b, a simple calculation shows that the expected number of adjacent comparisons globally is $O(1)$. This suggests that one needs $\Omega(n)$ blocks b to get all the $n-1$ adjacent comparisons, and hence a branching program of length $o(n^{5/3})$ would not be able to solve the element distinctness problem.

To carry out a proof along the above line, one needs to develop the arguments without the assumption that within each block b only variables among a small subset D_b can be used. Also, it seems that we might apply the argument recursively to the blocks b in the hope of getting lower bounds on TS better than $\Omega(n^{5/3})$. Theorem 1 is the result of the development of such a recursive proof.

Before presenting the proof, we introduce a concept important for all the ensuing technical discussions. Given any T inequalities $x_{i_1} < x_{j_1}, x_{i_2} < x_{j_2}, \ldots, x_{i_T} < x_{j_T}$ where $T \leq \frac{n}{2}$, let support (C) denote the set of all x_k involved in the inequalities, and let V be the set of the $2T - |\text{support}(C)|$ variables x_r with the smallest indices r that do not appear in support (C). Write support$'(C) = \text{support}(C) \cup V$. Let P_C denote the partial order generated by the T inequalities $x_{i_s} < x_{j_s}$ on support$'(C)$. The following proposition is obviously true.

Fact 1. Let C, D be two sequences of inequalities. If $C \subseteq D$, then support$'(C) \subseteq$ support$'(D)$.

We will use a measure of progress of the computation more refined than the one used in [BFMUW]. Roughly speaking, instead of using the number of adjacent comparisons among all x_i, we look at only the x_i we have seen so far and use the number of adjacent comparisons among them. We give an informal description here and will define it formally at the end of §2.2.

Let A be a branching program trying to solve the distinctness problem. How well is it doing, for example, after τ steps? Take a random linear ordering of n x_i as input, and let C be the sequence of comparisons encountered in the first τ steps. We concentrate on the subset support$'(C)$, and record the number of comparisons (call it κ) that are adjacent within support$'(C)$. We use the probability of $\kappa \geq m(\tau)$ (where $m(\tau)$ is some suitably chosen benchmark) as a measure of how well is A doing. If this probability is small, then A is not making good progress.

In the rest of this section, we introduce some additional needed notation and prove an auxiliary lemma. The proof of Theorem 1 will be given in §3.

2.2. Terminology. Let $W = \{w_1, w_2, \ldots, w_n\}$ be any nonempty finite set. A *linear ordering on* W is a sequence $\sigma = \langle w_{i_1} < w_{i_2} < \cdots < w_{i_n}\rangle$, in which each element of W appears exactly once. Let $\Gamma(W)$ denote the set of all linear orderings on W.

Let $P = (<_P, W)$ be a partial order on W. A linear ordering $\rho \in \Gamma(W)$ is said to be *consistent with* P, if $w <_P w'$ implies $w < w'$ in ρ. Let $\Delta(P)$ denote the set of all linear orderings on W consistent with P.

We will use the symbol X to denote exclusively the set $\{x_1, x_2, \ldots, x_n\}$ of the n input elements. Let $X' \subseteq X$. For any $\sigma \in \Gamma(X')$, let $L(\sigma)$ denote the set of all $\rho \in \Gamma(X)$ consistent with σ.

Suppose $\sigma = \langle x_{r_1} < x_{r_2} < \cdots < x_{r_l}\rangle$. A comparison $x_i < x_j$ is *adjacent in* σ if x_i, x_j are adjacent in the linear ordering σ, i.e., there exists an m such that $i = r_m$ and $j = r_{m+1}$. For any $\rho \in \Gamma(X)$, let $\rho|_{X'}$ denote the $\sigma \in \Gamma(X')$ obtained from the restriction of ρ to X'. For example, if $\rho = \langle x_3 < x_2 < x_5 < x_1 < x_4\rangle$ and $X' = \{x_2, x_4, x_5\}$, then $\rho|_{X'} = \langle x_2 < x_5 < x_4\rangle$.

Let C be a sequence of inequalities $(x_{i_1} < x_{j_1}, x_{i_2} < x_{j_2}, \ldots, x_{i_T} < x_{j_T})$. We say that C is *nontrivial* if there is a linear ordering $x_{r_1} < x_{r_2} < \cdots < x_{r_n}$ such that all inequalities in C are true. Let C be nontrivial and let $0 \leq a_1 < a_2 \leq T$ be two integers. For any $\sigma \in \Delta(P_C)$, let $Z(C, a_1, a_2, \sigma)$ be the number of pairs (i_s, j_s), $a_1 < s \leq a_2$, such that the comparisons $x_{i_s} < x_{j_s}$ are adjacent in σ.

Let A be any branching program of variables $x_1, x_2, \ldots, x_n$. For any node u and positive integer s, let $A[u, s]$ denote the subbranching program of A of length s and rooted at node u. A *path* δ in A is a sequence $u_0, e_1, u_1, e_2, \ldots, u_{s-1}, e_s, u_s$, where each e_r is an edge from node u_{r-1} to u_r; s is the *length* of δ. We will restrict ourselves to paths for which no edge is labeled by "=." Let C_δ denote the sequence of comparison inequalities of the form $x_i < x_j$ obtained along the path δ. For any linear ordering $\rho \in \Gamma(X)$, let $\tilde{x}_\rho$ denote an input $(x_1, x_2, \ldots, x_n)$ that satisfies all the inequalities in ρ. Let $\xi_{A,\rho}$ be the (entire) path traversed by input $\tilde{x}_\rho$ in A.

Let A be a branching program of length T. Let m be any integer. Take a random ρ uniformly chosen from $\Gamma(X)$. Let $g_A(m)$ denote the probability of the event $Z(C_\delta, 0, T, \rho|_{X'}) \geq m$, where $\delta = \xi_{A,\rho}$ and $X' = \text{support}'(C_\delta)$. We regard $g_A(m)$, with a suitably chosen m, as a measure of how effective A is in solving the distinctness problem. The plan for proving

Theorem 1 is to show that $g_A(m)$ is small unless T is sufficiently large; m is chosen in a way that accommodates an inductive proof.

2.3. An auxiliary lemma. We begin with an informal description of the problem to be addressed by the lemma. Let a subset $W \subseteq X$ of elements obeying a linear ordering σ on W be given, say, $x_{r_1} < x_{r_2} < \cdots < x_{r_{|W|}}$. Among the $|W|-1$ adjacent pairs $(x_{r_j}, x_{r_{j+1}})$, l of them are *marked*. Assume that it is known that all possible linear orderings $\rho \in \Gamma(X)$ consistent with σ are equally likely. We would like to find, by no more than T adaptive comparisons, a set of $2T$ elements outside W with the objective of maximizing the probability p of success, where *success* means that at least m of the marked adjacent pairs remain adjacent within the larger "output" set $W'(W'$ being the union of W plus the $2T$ elements found). The lemma provides an upper bound on the achievable p.

To specify the problem precisely, let $W \subseteq X$ be a nonempty set, and $\sigma \in \Gamma(W)$. Let C be a nontrivial sequence of comparisons $x_i < x_j$ with $x_i, x_j \in W$, with exactly l of them being adjacent in σ, where $0 \leq l < |W|$. Let A be a branching program of length T with $n \geq 2T + |W|$. For any possibly traversed path δ of length T, let W_δ be any subset of X such that $|W_\delta| = 2T + |W|$ and $W_\delta \supseteq W \cup \text{support}(C_\delta)$. Denote by $\mathcal{V}$ the collection $\{W_\delta |$ all paths $\delta\}$.

Now, take a random $\rho \in L(\sigma)$. Let $f_{n,A,W,\mathcal{V}}(\sigma, C, m)$ be the probability that the number of comparisons of C adjacent in $\rho|_{W_\delta}$ is greater or equal to m, where $\delta = \xi_{A,\rho}$.

LEMMA 1. $f_{n,A,W,\mathcal{V}}(\sigma, C, m) \leq \binom{l}{m} \cdot (|W|/2T)^m$.

Proof. Without loss of generality, assume that $W = \{x_1, x_2, \ldots, x_w\}$ where $w = |W|$, and σ is the linear ordering $x_1 < x_2 < \cdots < x_w$. Let C' be the set of the comparisons of C that are adjacent in σ, for example, $C' = \{x_{i_1} < x_{i_1+1}, x_{i_2} < x_{i_2+1}, \ldots, x_{i_l} < x_{i_l+1}\}$. Let $s = 2T + w$.

We express $f_{n,A,W,\mathcal{V}}$ in terms of a stochastic process. Take a random $\rho \in L(\sigma)$ and traverse the path $\xi_{A,\rho}$. Let us keep a sorted list W'; initially, W' is the sorted version of W. When we encounter a new node u with a comparison $x_r : x_{r'}$, we insert the elements in $\{x_r, x_{r'}\} - W'$, if any, one at a time into the ordered list W'. Note that each new element, when added to an ordered list of c elements, will be equally likely in any of the $c+1$ ranks. When we reach the leaf, we add $s - |W'|$ new elements of W_δ, one at a time, into W'. Again, each new element is equally likely to occupy any of the ranks currently possible in W'. The quantity $f_{n,A,W,\mathcal{V}}(\sigma, C, m)$ can thus be calculated as follows. We start with an ordered list of w items with l of the intervals (between the i_j and the $i_j + 1$st items for $1 \leq j \leq l$) marked; then we sequentially insert new items into the list. Each time the new item is equally likely to be inserted into any of the existing intervals; $f_{n,A,W,\mathcal{V}}(\sigma, C, m)$ is the probability that, after $2T$ insertions, at least m of the original l marked intervals remain intact (no item has been inserted into these intervals).

We will obtain an upper bound on $f_{n,A,W,\mathcal{V}}(\sigma, C, m)$. (Essentially, this is now reduced to a calculation that was done in [BFMUW].) Let us describe the process using a sequence of $2T$ integers $j_1, j_2, \ldots, j_{2T}$, where $1 \leq j_r \leq w + r$ is the rank of the rth inserted item when it is being inserted. Thus, there are in all $\prod_{1 \leq r \leq 2T}(w+r)$ configurations. To specify a configuration for which at least m marked intervals remain intact, we first specify m such intervals, and then specify the ranks of the inserted items by integers $j_1, j_2, \ldots, j_{2T}$, where $1 \leq j_r \leq w + r - m$. The total number of such configurations is thus at most $\binom{l}{m} \prod_{1 \leq r \leq 2T}(w + r - m)$. It follows that

$$f_{n,A,W,\mathcal{V}}(\sigma, C, m) \leq \binom{l}{m} \frac{\prod_{1 \leq r \leq 2T}(w + r - m)}{\prod_{1 \leq r \leq 2T}(w + r)},$$

$$= \binom{l}{m} \frac{w!\ (s-m)!}{s!\ (w-m)!},$$
$$= \binom{l}{m} \frac{w(w-1)\dots(w-m+1)}{s(s-1)\dots(s-m+1)},$$
$$\le \binom{l}{m} \cdot \left(\frac{w}{2T}\right)^m. \quad \square$$

Remarks. The bounds derived and the formulas involved in the above proof do not have any explicit dependency on the choice of W_δ. This is not a surprise, since it is not hard to see that $f_{n,A,W,\mathcal{V}}(\sigma, C, m)$ is in fact independent of $\mathcal{V}$. For any input consistent with σ, when we have carried out the comparisons according to A (traversing a path δ), the x_i's outside $W \cup$ support(C_δ) are isolated elements in the partial order generated by σ and the comparisons. The choice of W_δ only affects which of these isolated elements are to be included in the final *output* W' (in the sense described in the first paragraph of this subsection). By symmetry, this choice does not affect the probability of "success" (in the sense defined in the first paragraph of this subsection).

3. Proof of Theorem 1. The heart of the proof is the next lemma. Let $N_0 = 10^{14}, n \ge N_0, S > 0, t = \lceil e^{(\ln n)^{1/2}} \rceil$, and $k_0 = \lfloor \log_t(n/4) \rfloor$. Then $t > e^5, k_0 > 3$. For any integer $k > 0$, let $m_k = 2^{k_0+4k+16} t S$, and $q_k = 4^{(k-k_0)S}(4t)^{k-k_0} 2^{-10S}$.

MAIN LEMMA. *Let* $1 \le k \le k_0$. *Then* $g_A(m_k) \le q_k$, *for any branching program* A *of length* t^k *and capacity* S.

COROLLARY. *Let* A *be a branching program of length* $\le t^{k_0}$ *and capacity* S. *Take a random* ρ, *uniformly chosen from* $\Gamma(X)$ *and input* $\tilde{x}_\rho$ *to* A. *Then the probability that* A *makes at least* m_k *comparisons adjacent in* ρ *is* $\le q_k$.

The corollary follows from the Main Lemma, since the introduction of additional elements x_i into support$'(C_\delta)$ will not increase the number of adjacent comparisons.

We first restate the Main Lemma. Take a random ρ, uniformly chosen from $\Gamma(X)$, and input $\tilde{x}_\rho$ to A. Let $\delta = \xi_{A,_\rho}$. The Main Lemma asserts that

$$\Pr\{Z(C_\delta, 0, t^k, \rho|_{X'}) \ge m_k\} \le q_k,$$

where $X' = \text{support}'(C_\delta)$.

We will prove the above inequality by induction on $k \ge 1$. For $k = 1$, we have $m_k > t^k$. Since A can make only t^k comparisons, the inequality is true (the left-hand side being 0).

We now assume that $1 < k \le k_0$, and that the Main Lemma has been proved for all values less than k.

Let $\rho \in \Gamma(X)$. Write $\delta = \xi_{A,_\rho}$ and $X' = \text{support}'(C_\delta)$. If $Z(C_\delta, 0, t^k, \rho|_{X'}) \ge m_k$, then there is a $1 \le d_\rho \le t$, such that $Z(C_\delta, (d_\rho - 1)t^{k-1}, d_\rho t^{k-1}, \rho|_{X'}) \ge m_k/t$. Thus, for a random ρ uniformly chosen from $\Gamma(X)$,

$$(1) \quad \Pr\{Z(C_\delta, 0, t^k, \rho|_{X'}) \ge m_k\} \le \sum_{1 \le d \le t} \Pr\{Z(C_\delta, (d-1)t^{k-1}, dt^{k-1}, \rho|_{X'}) \ge m_k/t\}.$$

We will show that, for each $1 \le d \le t$,

$$(2) \qquad \Pr\{Z(C_\delta, (d-1)t^{k-1}, dt^{k-1}, \rho|_{X'}) \ge m_k/t\} \le (2^s + 1)q_{k-1}.$$

This will complete the inductive proof of the Main Lemma, since it follows from (1) and (2) that

$$\Pr\{Z(C_\delta, 0, t^k, \rho|_{X'}) \ge m_k\} \le (2^S + 1)tq_{k-1}$$
$$\le q_k.$$

Fix $1 \leq d \leq t$. Let $v_1, v_2, \ldots, v_r$ be the nodes of A at level $(d-1)t^{k-1}$. For each i, let B_i be the set of paths β of length t^{k-1} starting at node v_i. Let

$$\Psi_i = \cup_{\beta \in B_i} \Delta(P_{C_\beta}).$$

Note that Ψ_i consists of all linear orderings σ of the comparisons along β restricted to the subset support$'(C_\beta)$ (not just support(C_β)) for all paths $\beta \in B_i$. The following two facts are obviously true. Let $1 \leq i \leq r$.

Fact 2. Let $\beta, \beta' \in B_i$. If $\sigma \in \Delta(P_{C_\beta}) \cap \Delta(P_{C_{\beta'}})$, then $\beta = \beta'$. We can thus write $\beta(i, \sigma)$ for the unique $\beta \in B_i$ for which $\sigma \in \Delta(P_{C_\beta})$.

Fact 3. The family $L(\sigma)$, $\sigma \in \Psi_i$, forms a partition of the set $\Gamma(X)$.

To prove (2), we divide the set of inputs according to which node v_i an input reaches. For $1 \leq i \leq r$, let R_i be the set of $\rho \in \Gamma(X)$ such that input $\tilde{x}_\rho$ will reach v_i A. We further subdivide each R_i into three parts. Let $\Psi_i^{(1)}$ be the set of $\sigma \in \Psi_i$ such that $C_{\beta(i,\sigma)}$ contains at least m_{k-1} comparisons adjacent in σ. Let $\Psi_i^{(2)}$ be the set of $\sigma \in \Psi_i$ such that $|L(\sigma) \cap R_i| \leq |L(\sigma)| q_{k-1}/(10 \cdot 2^S)$. Let $\Psi_i^{(0)} = \Psi_i - \Psi_i^{(1)} - \Psi_i^{(2)}$. We will estimate the contribution to $\Pr\{Z(C_\delta, (d-1)t^{k-1}, dt^{k-1}, \rho|_{X'}) \geq m_k/t\}$ from

$$\rho \in R_i \cap (\cup_{\sigma \in \Psi_i^{(j)}} L(\sigma))$$

for each $j \in \{0, 1, 2\}$. The overwhelming majority of the ρ's belong to the part $j = 0$; the part $j = 2$ groups some infrequently visited nodes and paths; the part $j = 1$ gathers paths making many local adjacent comparisons, and hence cannot include too many ρ's by the induction hypothesis. Facts 4–6 give estimates of the contributions from the various parts to the right-hand side of (2).

Generate a random ρ uniformly chosen from $\Gamma(X)$. Let δ be $\xi_{A,\rho}$, the path traversed by $\tilde{x}_p$ in A. We consider the probabilities for two events. First, let $p_{i,\sigma}$ be the probability that $\rho \in L(\sigma)$ and the path δ contains v_i. Clearly, $p_{i,\sigma} = |R_i \cap L(\sigma)|/n!$. Second, let $\alpha_{i,\sigma}$ be the conditional probability that, for such an input $\tilde{x}_\rho$, the branching program $A[v_i, t^{k-1}]$ will make m_k/t comparisons adjacent in $\rho|_{X'}(X' = \text{support}'(C_\delta))$. Clearly,

$$\begin{aligned}(3)\qquad &\Pr\{Z(C_\delta, (d-1)t^{k-1}, dt^{k-1}, \rho|_{X'}) \geq m_k/t\} \\ &\qquad = \sum_{1 \leq i \leq r} \sum_{\sigma \in \Psi_i} p_{i,\sigma}\alpha_{i,\sigma}.\end{aligned}$$

We need three facts. Let $1 \leq i \leq r$.

Fact 4.

$$\sum_{\sigma \in \Psi_i^{(1)}} p_{i,\sigma} \leq q_{k-1}.$$

Fact 5.

$$\sum_{\sigma \in \Psi_i^{(2)}} p_{i,\sigma} \leq q_{k-1}/(10 \cdot 2^S).$$

Fact 6. For each $\sigma \in \Psi_i^{(0)}$, $\alpha_{i,\sigma} \leq q_{k-1}/10$.

Applying the induction hypothesis to $A[v_i, t^{k-1}]$, we obtain

$$\frac{1}{n!}\left|\cup_{\sigma \in \Psi_i^{(1)}} L(\sigma)\right| \leq q_{k-1},$$

from which Fact 4 follows immediately. We obtain Fact 5 from the following derivation, using Fact 3 in the last step,

$$\begin{aligned}\sum_{\sigma\in\Psi_i^{(2)}} p_{i,\sigma} &= \sum_{\sigma\in\Psi_i^{(2)}} \frac{|L(\sigma)\cap R_i|}{n!}\\ &= \frac{1}{n!}\sum_{\sigma\in\Psi_i^{(2)}} |L(\sigma)|\frac{|L(\sigma)\cap R_i|}{|L(\sigma)|}\\ &\le \frac{1}{n!}\frac{1}{10\cdot 2^S}q_{k-1}\sum_{\sigma\in\Psi_i^{(2)}}|L(\sigma)|\\ &\le \frac{1}{10\cdot 2^S}q_{k-1}.\end{aligned}$$

We now prove Fact 6. Let $\sigma\in\Psi_i^{(0)}$. Let u denote the node at the end of the path $\beta(i,\sigma)$. Thus u is at a distance dt^{k-1} from the source of the branching program A.

We first construct a new branching program $A_{i,\sigma}$. Take the branching program $A[\text{root}, (d-1)t^{k-1}]$, and attach to the end node v_i a copy of $A[u, t^k - dt^{k-1}]$ by identifying u with v_i. This is now a (nonleveled) branching program that may also be obtained by deleting from A all outgoing edges from all nodes v_j at level $(d-1)t^{k-1}$ except the one edge (out of v_i) on the path $\beta(i,\sigma)$, contracting the path $\beta(i,\sigma)$ to a point, and pruning away nodes and edges unreachable by inputs. Call this the *main portion* of $A_{i,\sigma}$. We finish the construction by adding arbitrary pieces to the other end nodes $v_j (j\neq i)$ to make it a leveled branching program (to conform to our convention of the definition of a branching program); for example, for each $v_j\neq v_i$, we can sequentially make the comparisons $x_a : x_{a+1}$ for $a = 1, 2, \ldots, t^k - dt^{k-1}$ and go to new nodes $u_{j,a}$ irrespective of the results of the comparisons. Clearly, $A_{i,\sigma}$ is of length $t^k - t^{k-1}$. Note that the capacity of $A_{i,\sigma}$ depends on how we choose the pieces outside the main portion and may be different from the capacity of A. We will use $A_{i,\sigma}$ only for the purpose of analyzing $\alpha_{i,\sigma}$.

For any path δ of length t^k in A starting at the source and containing $\beta(i,\sigma)$ there is an obvious corresponding path $\hat{\delta}$ of length $t^k - t^{k-1}$ in $A_{i,\sigma}$ that lies completely in the main portion. The converse is also true. Any such $\hat{\delta}$ in $A_{i,\sigma}$ will be called a *central path*.

For any central path $\hat{\delta}$ in $A_{i,\sigma}$, define $W_{\hat{\delta}} = \text{support}'(C_\delta)$. For any noncentral path δ' of length $t^k - t^{k-1}$ in $A_{i,\sigma}$, choose $W_{\delta'}$ arbitrarily but subject to the conditions $|W_{\delta'}| = 2t^k$ and $\text{support}'(C_{\delta'})\cup \text{support}'(C_{\beta(i,\sigma)}) \subseteq W_{\delta'}$. This is always feasible since $|\text{support}'(C_{\delta'})| = 2(t^k - t^{k-1})$ and $|\text{support}'(C_{\beta(i,\sigma)})| = 2t^{k-1}$. (We use the notation δ' for paths in $A_{i,\sigma}$ to avoid confusion with the paths in A.) Let $\mathcal{V}$ be the collection of $W_{\delta'}$ for all paths δ' (of length $t^k - t^{k-1}$) in $A_{i,\sigma}$.

Note that, for any central path $\hat{\delta}$ in $A_{i,\sigma}$, we have $W_{\hat{\delta}} = \text{support}'(C_\delta)$ by definition. Also from Fact 1 (§2.1) $\text{support}'(C_\delta)$ contains both $\text{support}'(C_{\beta(i,\sigma)})$ and $\text{support}'(C_{\hat{\delta}})$; this implies $\text{support}'(C_{\beta(i,\sigma)})\cup \text{support}'(C_{\hat{\delta}}) \subseteq W_{\hat{\delta}}$. Thus, for all paths δ' (of length $t^k - t^{k-1}$) in $A_{i,\sigma}$,

$$\text{support}'(C_{\beta(i,\sigma)}) \cup \text{support}'(C_{\delta'}) \subseteq W_{\delta'}. \tag{4}$$

Let b denote the probability that, for a random $\tilde{x}_\rho$, with ρ uniformly chosen from $R_i\cap L(\sigma)$, the path δ' traversed by $\tilde{x}_\rho$ in the branching program $A_{i,\sigma}$ possesses the property that $C_{\beta(i,\sigma)}$ contains at least m_k/t comparisons adjacent in $W_{\delta'}$. (Note that $\beta(i,\sigma)$ is *not* a subpath of δ' but the path in A between nodes v_i and u.) Define c in the same fashion as b was defined, except that ρ is uniformly chosen from $L(\sigma)$. (These are well defined, since support $(C_{\beta(i,\sigma)}) \subseteq W_{\delta'}$ by (4).) Because any ρ satisfying the defining constraints for b must satisfy the defining constraints for c, we have

$$b\cdot|R_i\cap L(\sigma)| \le c\cdot|L(\sigma)|. \tag{5}$$

Let $\rho \in R_i \cap L(\sigma)$. If $\tilde{x}_\rho$ is input to A and traverses a path δ, then the same $\tilde{x}_\rho$ when input to $A_{i,\sigma}$ will trace the central path $\hat{\delta}$; furthermore, by definition we have $W_{\hat{\delta}} = \mathrm{support}'(C_\delta)$. Recall that $\alpha_{i,\sigma}$ is the probability that, for a random $\tilde{x}_\rho$, with ρ uniformly chosen from $R_i \cap L(\sigma)$, $C_{\beta(i,\sigma)}$ contains at least m_k/t comparisons adjacent in $\rho|'_X(X' = \mathrm{support}'(C_\delta)$ with $\delta = \xi_{A,\rho})$. Comparing this with the definition of b, we have

$$\alpha_{i,\sigma} = b. \tag{6}$$

Now, let $W = \mathrm{support}'(C_{\beta(i,\sigma)})$ and $T = t^k - t^{k-1}$; then $|W| = 2t^{k-1}$ and $2T + |W| = 2t^k$. Note that $n \geq 4t^k \geq 2T + |W|$. From (4), we have $W_{\delta'} \supseteq W \cup \mathrm{support}(C_{\delta'})$ for all paths δ' (of length T) in $A_{i,\sigma}$. From the definition of $f_{n,A,W,\mathcal{V}}$ stated at the beginning of §2.3, we see that c is exactly equal to $f_{n,A_{i,\sigma},W,\mathcal{V}}(\sigma, C_{\beta(i,\sigma)}, \lceil m_k/t \rceil)$. We have thus from (5) and (6) and the fact that $\sigma \in \Psi_i^{(0)}$

$$\begin{aligned} \alpha_{i,\sigma} &\leq \frac{|L(\sigma)|}{|R_i \cap L(\sigma)|} \cdot c \\ &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot f_{n,A_{i,\sigma},W,\mathcal{V}}(\sigma, C_{\beta(i,\sigma)}, \lceil m_k/t \rceil). \end{aligned} \tag{7}$$

Let l_σ denote the number of comparisons of $C_{\beta(i,\sigma)}$ adjacent in σ. Then by definition $l_\sigma \leq m_{k-1}$ for any $\sigma \in \Psi_i^{(0)}$. By applying Lemma 1 to the branching program $A_{i,\sigma}$ to get an upper bound on $f_{n,A_{i,\sigma},W,\mathcal{V}}(\sigma, C_{\beta(i,\sigma)}, \lceil m_k/t \rceil)$, and then substituting it into (7), we obtain

$$\begin{aligned} \alpha_{i,\sigma} &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot \binom{l_\sigma}{\lceil m_k/t \rceil} \left(\frac{2t^{k-1}}{2(t^k - t^{k-1})} \right)^{\lceil m_k/t \rceil} \\ &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot \binom{m_{k-1}}{\lceil m_k/t \rceil} (t-1)^{-\lceil m_k/t \rceil} \\ &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot \left(\frac{e m_{k-1}}{(t-1)\lceil m_k/t \rceil} \right)^{\lceil m_k/t \rceil} \\ &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot \left(\frac{et}{16(t-1)} \right)^{m_k/t} \\ &\leq \frac{10 \cdot 2^S}{q_{k-1}} \cdot \left(\frac{1}{2} \right)^{2^{16+k_0}S} \\ &\leq \frac{1}{10} q_{k-1}. \end{aligned}$$

This proves Fact 6.

We will now complete the proof of (2). From (3) and Facts 4–6, we obtain $Pr\{Z(C_\delta, (d-1)t^{k-1}, dt^{k-1}, \rho|_{X'}) \geq m_k/t\}$

$$\begin{aligned} &= \sum_{1 \leq i \leq r} \sum_{\sigma \in \Psi_i} p_{i,\sigma} \alpha_{i,\sigma} \\ &\leq \sum_i \sum_{\sigma \in \Psi_i^{(1)}} p_{i,\sigma} + \sum_i \sum_{\sigma \in \Psi_i^{(2)}} p_{i,\sigma} \\ &\quad + \sum_i \sum_{\sigma \in \Psi_i^{(0)}} p_{i,\sigma} \alpha_{i,\sigma} \end{aligned}$$

$$
\begin{aligned}
&\le q_{k-1}\sum_i 1 + \sum_i \frac{1}{10\cdot 2^S} q_{k-1} + \sum_i \sum_{\sigma\in\Psi_i^{(0)}} p_{i,\sigma}\frac{1}{10} q_{k-1}\\
&\le 2^S q_{k-1} + \frac{1}{10} q_{k-1} + \frac{1}{10} q_{k-1}\\
&\le (2^S+1) q_{k-1}.
\end{aligned}
$$

We have proved (2). This completes the inductive step in the proof of the Main Lemma.

We will now use the Main Lemma to complete the proof of Theorem 1. Let $n \ge N_0$, where $N_0 = 10^{14}$. Because $k_0 \le \ln n/\ln t \le (\ln n)^{1/2}$, we have

$$
\begin{aligned}
t\cdot 2^{5k_0} &\le 2\cdot \exp((\ln n)^{1/2} + k_0 5\ln 2)\\
&= O\left(n^{\epsilon(n)}\right).
\end{aligned} \tag{8}
$$

Let $A \in \mathcal{A}_n$. If $S \ge \frac{1}{10}\frac{n}{t}2^{-5k_0-16}$, then since $T \ge n-1$, we have by (8) $TS = \Omega(n^2/t2^{5k_0}) = \Omega(n^{2-\epsilon(n)})$. We now consider the case $S < \frac{1}{10}\frac{n}{t}2^{-5k_0-16}$. In this case

$$
0 < m_{k_0} < n/10. \tag{9}
$$

Since A has to have at least $n-1$ nodes, we have $S \ge \log_2(n-1) > 0$. Suppose that A has length $T < t^{k_0}n/2m_{k_0}$. We will derive a contradiction.

Let $\rho \in \Gamma(X)$. For each node v, let A_v denote the branching program $A[v, \min\{t^{k_0}, T - l_v\}]$, where l_v is the level number of v. Let K_v be the set of $\rho \in \Gamma(X)$ such that $C_{\delta(v,\rho)}$ contains at least m_{k_0} comparisons adjacent in ρ, where $\delta(v,\rho) = \xi_{A_v,\rho}$. By the corollary to the Main Lemma, $|K_v| \le q_{k_0}\cdot n!$.

Thus,

$$
\begin{aligned}
|\cup_v K_v| &\le 2^S q_{k_0}\cdot n!\\
&< n!.
\end{aligned}
$$

It follows that there exists a $\rho \notin \cup_v K_v$. Let us input $\tilde{x}_\rho$ to A. The traversed path δ can be decomposed into $\delta(v_1,\rho)\delta(v_2,\rho)\ldots\delta(v_d,\rho)$, where $d = \lceil T/t^{k_0}\rceil$, v_1 is the source, and $v_1, v_2, \ldots, v_d$ are nodes in δ with distance t^{k_0} between v_i and v_{i+1}; for each i, the subpath $\delta(v_i,\rho)$ is a path in the subbranching program A_{v_i}. For each i, as $\rho \notin K_{v_i}$, the number of comparisons made along $\delta(v_i,\rho)$ contains less than m_{k_0} comparisons adjacent in ρ. It follows that the total number of comparisons made by A that are adjacent in ρ is less than $dm_{k_0} \le \lceil n/2m_{k_0}\rceil\cdot m_{k_0} \le m_{k_0} + \frac{n}{2}$, which by (9) is less than $n-1$. This is a contradiction.

Thus, using (8) we obtain

$$
\begin{aligned}
T &\ge t^{k_0}\frac{n}{2m_{k_0}}\\
&= \Omega\left(\frac{t^{k_0-1}}{2^{5k_0}}\frac{n}{S}\right)\\
&= \Omega\left(\frac{n^2}{S}\frac{1}{t2^{5k_0}}\right)\\
&= \Omega\left(\frac{1}{S}n^{2-\epsilon(n)}\right).
\end{aligned}
$$

This completes the proof of Theorem 1.

4. Concluding remarks. In this paper we have shown a $TS = \Omega(n^{2-\epsilon(n)})$ time-space tradeoff for the element distinctness problem for comparison branching programs. It remains an interesting open question whether $TS = \Omega(n^2)$. Another direction of research is to investigate whether the time-space tradeoff presented here can be obtained for linear-test or arithmetic branching programs.

Acknowledgments. The author thanks Paul Beame for pointing out an error in the statement of Fact 4 in the original draft of this paper. The author also wishes to thank an anonymous referee for valuable suggestions to correct an inaccuracy and improve the presentations.

REFERENCES

[B] P. Beame, *A general sequential time-space tradeoff for finding unique elements*, SIAM J. Comput., 20 (1991), pp. 270–277.

[BC] A. Borodin and S. Cook, *A time-space tradeoff for sorting on a general sequential model of computation*, SIAM J. Comput., 11 (1982), pp. 287–297.

[BFMUW] A. Borodin, F. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson, *A time-space tradeoff for element distinctness*, SIAM J. Comput., 16 (1987), pp. 97–99.

[BFKLT] A. Borodin, M. Fischer, D. Kirkpatrick, N. Lynch, and M. Tompa, *A time-space tradeoff for sorting on oblivious machines*, J. Comput. System Sci., 22 (1981), pp. 351–364.

[C] A. Cobham, *The recognition problem for the set of perfect squares*, Research paper RC-1704, IBM Watson Research Center, Yorktown Heights, NY, April 1966.

[DG] P. Duris and Z. Galil, *A time-space tradeoff for language recognition*, Math. Systems Theory, 17 (1984), pp. 3–12.

[K] M. Karchmer, *Two time-space tradeoffs for element distinctness*, Theoret. Comput. Sci., 47 (1986), pp. 237–246.

[MP] J. I. Munro and M. Paterson, *Selection and sorting with limited storage*, Proceedings of Nineteenth IEEE Symposium on Foundations of Computer Science, Ann Arbor, Michigan, October 1978, pp. 253–258.

[Th] C. D. Thompson, *Area-time complexity for VLSI*, Proceedings Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, Georgia, May 1979, pp. 81–88.

[To] M. Tompa, *Time-space tradeoffs for computing functions using connectivity properties of their circuits*, J. Comput. System Sci., 20 (1980), pp. 118–132.

[U] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, 1984.

[Y] A. C. Yao, *On the time-space tradeoff for sorting with linear queries*, Theoret. Comput. Sci., 19 (1982), pp. 203–218.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 976–989, October 1994

006

EXISTENCE AND CONSTRUCTION OF EDGE-DISJOINT PATHS ON EXPANDER GRAPHS*

ANDREI Z. BRODER†, ALAN M. FRIEZE‡ AND ELI UPFAL§

Abstract. Given an expander graph $G = (V, E)$ and a set of q disjoint pairs of vertices in V, the authors are interested in finding for each pair (a_i, b_i) a path connecting a_i to b_i such that the set of q paths so found is edge disjoint. (For general graphs the related decision problem is NP complete.)

The authors prove sufficient conditions for the existence of edge-disjoint paths connecting any set of $q \leq n/(\log n)^\kappa$ disjoint pairs of vertices on any n vertex bounded degree expander, where κ depends only on the expansion properties of the input graph, and not on n. Furthermore, a randomized $o(n^3)$ time algorithm, and a random $\mathcal{NC}$ algorithm for constructing these paths is presented. (Previous existence proofs and construction algorithms allowed only up to n^ϵ pairs, for some $\epsilon \ll \frac{1}{3}$, and strong expanders [D. Peleg and E. Upfal, Combinatorica, 9 (1989), pp. 289–313.].)

In passing, an algorithm is developed for splitting a sufficiently strong expander into two edge-disjoint spanning expanders.

Key words. edge-disjoint paths, expanders

AMS subject classifications. 05C85, 05C38, 68Q20, 68R10, 60J15, 90B10, 90B12

1. Introduction. Given an expander graph $G = (V, E)$ and a set of q disjoint pairs of vertices in V, we are interested in finding for each pair (a_i, b_i), a path connecting a_i to b_i, such that the set of q paths so found is edge disjoint.

For arbitrary graphs, the related decision problem is in $\mathcal{P}$ for fixed q [21], but is $\mathcal{NP}$ complete if q is part of the input. However, this negative result can be circumvented for certain classes of graphs. For certain bounded degree expander graphs, Peleg and Upfal [19] have presented a polynomial time algorithm and a random $\mathcal{NC}$ algorithm for constructing up to n^ϵ disjoint paths, where $0 < \epsilon \ll \frac{1}{3}$ is a constant that depends on the expansion properties of the input graph.

In this paper we describe a new algorithm for constructing edge-disjoint paths. Using it we can construct up to $n/(\ln n)^\kappa$ disjoint paths on bounded degree graphs with a sufficiently strong expansion property, where κ is a constant that depends only on the expansion property of the input graph. Our algorithm is based on two probabilistic tools: the rapid mixing properties of random walks on expanders and the Lovász Local Lemma [10].

As in [19], the disjoint paths are constructed in two stages. In the first stage we choose a random set Q of $2q$ vertices that are at least $\kappa_1 \ln \ln n$ apart from each other. (The constant κ_1 will be defined later.) We connect the original endpoints to the vertices of Q in an arbitrary fashion via edge-disjoint paths, such that each Q-vertex is the endpoint of exactly one path. A simple flow argument proves constructively the existence of such edge-disjoint paths on any graph with edge expansion larger than one.

Let $\tilde{a}_i$ (respectively, $\tilde{b}_i$) denote the vertex in Q that was connected to the original endpoint a_i (respectively, b_i) in the first stage. The core of the algorithm consists of constructing edge-disjoint paths connecting $\tilde{a}_i$ to $\tilde{b}_i$, for $i = 1, \ldots, q$.

In the second stage of the algorithm we choose for each pair $(\tilde{a}_i, \tilde{b}_i)$ a bundle of $(\ln n)^2$ random paths connecting them. The goal is to show via the Lovász Local Lemma that there

*Received by the editors June 1, 1992; accepted for publication (in revised form) May 11, 1993.

†DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301.

‡Department of Mathematics, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15218. A portion of this author's work was done while the author was visiting DEC Systems Research Center and was supported in part by National Science Foundation grants CCR8900112 and CCR9024935.

§IBM Almaden Research Center, San Jose, California 95120, and Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

is one choice out of each bundle of paths such that the set of chosen paths is edge disjoint. However, the Local Lemma cannot be applied to sets of paths chosen uniformly at random, since the dependency graph that corresponds to such choices is too dense. The crux of the proof is that for any expander graph there is a simple pruning mechanism that reduces the dependency so that the Lemma can be applied.

To convert the existence proof to an explicit algorithm, we observe that the dependency graph constructed in this process is almost surely composed of sufficiently small components, such that the paths can be selected in polynomial time by exhaustive search.

Each of the two stages of the algorithm requires an expander graph. If we apply both stages to the same expander we prove the following theorem.

THEOREM 1.1. *Given any n vertex, bounded degree, regular graph G with edge expansion greater than one, and given any set of $q \leq n/(\log n)^{\kappa}$ disjoint pairs of vertices in G, with high probability our algorithm finds in $o(n^3)$ steps a set of paths in G connecting the q pairs, such that each edge in G participates in no more than two paths.* (*The constant κ is exposed in the proof.*)

While the above result is sufficient for most applications, it is still of theoretical interest to find edge-disjoint paths. To achieve this goal we develop a new algorithm for splitting a sufficiently strong expander into two edge-disjoint spanning expanders. We believe that this algorithm is of independent interest. The splitting algorithm requires a stronger but still bounded degree expander. Splitting it and applying each stage of our algorithm to a different set of edges we prove the following theorem.

THEOREM 1.2. *Given an n vertex graph with sufficiently strong edge expansion, and given any set of $q \leq n/(\log n)^{\kappa}$ disjoint pairs of vertices in G, with high probability our algorithm finds in $o(n^3)$ steps a set of edge-disjoint paths in G connecting the q pairs.* (*The constant κ and the required edge expansion are defined later.*)

The disjoint paths problem has numerous algorithmic applications. One that has received increased attention in recent years is in the context of communication networks for parallel and distributed computing. While packet routing is the communication protocol of choice for bounded size messages, it cannot always be used efficiently for high volume communication such as in multimedia applications and two-way communication. A more efficient way to transmit such information is through disjoint paths (virtual circuits) that are dedicated to one pair of processors for the duration of the communication. Our result gives yet more evidence of the usefulness of a communication network with strong expansion properties [24], [16], [20].

A preliminary version of this paper has appeared in [8].

2. Preliminaries.

There are various ways to define expander graphs; here we define them in terms of edge expansion (a weaker property than vertex expansion).

Let $G = (V, E)$ be a graph. For a set of vertices $S \subset V$ let out(S) be the set of edges with one endpoint in S and one endpoint in $V \setminus S$, that is

$$\text{out}(S) = \Big\{\{u, v\} \mid \{u, v\} \in E, u \in S, v \notin S\Big\}.$$

Similarly

$$\text{in}(S) = \Big\{\{u, v\} \mid \{u, v\} \in E, u \in S, v \in S\Big\}.$$

DEFINITION 1. A graph $G = (V, E)$ is a β-expander if for every set $S \subset V$, $|S| \leq |V|/2$ we have $|\text{out}(S)| \geq \beta|S|$.

DEFINITION 2. An r-regular graph $G = (V, E)$ is an (α, β, γ)-expander if for every set $S \subset V$

$$\text{out}(S) \geq \begin{cases} (3r/4 + \alpha)|S| & \text{if } |S| \leq \gamma|V|, \\ \beta|S| & \text{if } \gamma|V| < |S| \leq |V|/2. \end{cases}$$

DEFINITION 3. The t-neighborhood of a vertex u in G is the set of all vertices that are at distance t or less from u in G.

The notation $\text{dist}(u, v)$ refers to the distance from vertex v to vertex u. If U is a set of vertices, then $\text{dist}(U, v) = \min_{u \in U} \text{dist}(u, v)$. If P is a path (which we view as a set of edges), then $\text{dist}(P, v)$ is the minimum over all vertices u on P of $\text{dist}(u, v)$. These definitions generalize in the obvious manner when both arguments are paths, sets, and so on.

A *random walk* on the undirected graph $G = (V, E)$ is a Markov chain $\{X_t\} \subseteq V$ associated with a particle that moves from vertex to vertex according to the following rule: the probability of a transition from vertex i, of degree d_i, to vertex j is $1/d_i$ if $\{i, j\} \in E$, and 0 otherwise. (In case of a bipartite graph we need to assume that we do nothing with probability $\frac{1}{2}$ and move off with probability $\frac{1}{2}$ only. This technicality is ignored for the remainder of the paper.) Its stationary distribution, denoted π, (or $\pi(G)$) is given by $\pi_v = d_v/(2|E|)$. A trajectory W of length τ is a sequence of vertices $[w_0, w_1, \ldots, w_\tau]$ such that $\{w_t, w_{t+1}\} \in E$. The Markov chain $\{X_t\}$ induces a probability distribution on trajectories, namely the product of the probabilities of the transitions that define the trajectory.

The notation $B(m, p)$ stands for the binomial random variable with parameters m = number of trials, and p = probability of success.

The notation $[m]$ stands for the set $\{1, 2, \ldots, m\}$ for any positive integer m.

3. The sequential algorithm. Below we present the algorithm for finding disjoint paths. If we relax the requirements so that edges can be used twice, then the input can be any β-expander, $\beta > 1$, and phase 1 is omitted entirely. The algorithm involves certain constants $\kappa, \kappa_1, \kappa_2, \kappa_3$, and κ_4. The required relations between these constants are explicitly given in equations (13)–(15). At various points of the algorithm we stop if certain conditions fail to hold. The subsequent analysis shows that premature termination is unlikely.

Algorithm DisjPaths
Input: An r-regular graph $G = (V, E)$ with sufficiently strong expansion. A collection of q disjoint pairs of vertices $\{(a_1, b_1), \ldots, (a_q, b_q)\}$. (The term "sufficiently strong" will be fully explained below.)

Output: A set of q edge-disjoint paths $\{P_1, \ldots, P_q\}$ such that P_i connects a_i to b_i.

Phase 1. Split G into two spanning expanders $G_R = (V, E_R)$ and $G_B = (V, E_B)$ such that $E = E_R \cup E_B$ and $E_R \cap E_B = \emptyset$. We require G_R to be a 1-expander and G_B to be a β'-expander for some $\beta' > 0$. (The details of this procedure are presented in §4.1.)

The steady state distribution of the random walk on G_B is easily seen to be given by

$$\pi(v) = \frac{d_B(v)}{2|E_B|}, \quad v \in V,$$

where $d_B(v)$ denotes the degree of the vertex v in G_B. Our construction guarantees that

$$\frac{1}{2n} \leq \pi(v) \leq \frac{3}{2n} \quad \forall v \in V. \tag{1}$$

Phase 2. Choose independently (with replacement) according to the distribution $\pi(G_B)$, a multiset of $4q$ vertices in V. Let $R = \{r_1, \ldots, r_{4q}\}$ be the multiset of vertices so chosen.

Phase 3. Select a set $Q \subset R$ of $2q$ vertices, such that every pair of vertices in Q are $\kappa_1 \ln\ln n$ apart from each other, as follows.

```
$Q \leftarrow \emptyset$
for $i = 1, \ldots, 4q$ while $|Q| < 2q$ do
    if dist$(Q, r_i) \geq \kappa_1 \ln\ln n$ then $Q \leftarrow Q \cup \{r_i\}$ fi
od
```

If at the end of this procedure $|Q| < 2q$ then **stop**. The algorithm has failed.

Phase 4. Let $S = \{a_1, \ldots, a_q, b_1 \ldots, b_q\}$. Using a flow algorithm in G_R, connect in an arbitrary manner the vertices of S to the vertices of Q by $2q$ edge-disjoint paths. (Except for the edges on these paths, no other edges of G_R are used for the final construction.) If such a flow cannot be constructed then **stop**. The algorithm has failed. (This can happen only if G_R did not have sufficient edge expansion.)

Phase 5. Let $\tilde{a}_i$ (respectively, $\tilde{b}_i$) be the vertex in Q that was connected to a_i (respectively, b_i). For each pair $(\tilde{a}_i, \tilde{b}_i)$ construct $m = (\ln n)^2$ paths, $P_{i,1}, \ldots, P_{i,m}$ connecting $\tilde{a}_i$ to $\tilde{b}_i$, as follows.

```
for $j = 1, 2, \ldots, m$ do
    Pick a vertex $x_{i,j}$ according to the distribution $\pi(G_B)$.
    Choose a trajectory $W'_{i,j}$ (respectively, $W''_{i,j}$) of length
      $\tau = \kappa_2 \ln n$ that goes from $\tilde{a}_i$ to $x_{i,j}$ (respectively, $\tilde{b}_i$ to $x_{i,j}$)
      in $G_B$, according to the distribution on trajectories,
      conditioned on $w_{i,j,0} = \tilde{a}_i$ and $w_{i,j,\tau} = x_{i,j}$. (The
      distribution for $W''_{i,j}$ is analogous.)
    Let $W_{i,j}$ be the walk formed by $W'_{i,j}$ followed by $W''_{i,j}$ reversed.
    Reduce $W_{i,j}$ to a path $P_{i,j}$ by removing cycles.
od
```

(The purpose of the remainder of the algorithm is to find among the set of $q \cdot m$ paths constructed in this phase a *solution set*, that is, a subset of q edge-disjoint paths, one for each pair $(\tilde{a}_i, \tilde{b}_i)$.)

Phase 6. We refer to the set of paths $B_i = \{P_{i,1}, P_{i,2}, \ldots, P_{i,m}\}$ as *bundle i*. The purpose of this phase is to prune from each bundle those paths that go "too close" to the endpoints of other bundles or to each other.

Let $w'_{i,j,t}$ and $w''_{i,j,t}$ denote the tth vertices of $W'_{i,j}$ and $W''_{i,j}$, respectively. Let $M_{i,j} = \{w'_{i,j,t}, w''_{i,j,t} : t \geq (\kappa_1 - \kappa_3) \ln\ln n\}$.

```
for $i = 1, 2, \ldots, q$ do
    for $j = 1, 2, \ldots, m$ do
        (a) if dist$(M_{i,j}, \bigcup_{k<j} M_{i,k}) \leq 2\kappa_3 \ln\ln n$ then
            $B_i \leftarrow B_i \setminus \{P_{i,j}\}$ fi
        (b) if dist$(W_{i,j}, Q \setminus \{\tilde{a}_i, \tilde{b}_i\}) < \kappa_3 \ln\ln n$ then
            $B_i \leftarrow B_i \setminus \{P_{i,j}\}$ fi
    od
od
```

(Condition (a) ensures that outside the $(\kappa_1 - \kappa_3) \ln\ln n$ neighborhood of the common endpoints, all paths remaining in B_i are at least $2\kappa_3 \ln\ln n$ apart. Condition (b) ensures that all paths in B_i are at least $\kappa_3 \ln\ln n$ from the endpoints of other bundles.)

Let m_i denote the number of paths left in bundle i for $i = 1, 2, \dots, q$, and rename the paths such that $B_i = \{P_{i,1}, \dots, P_{i,m_i}\}$.

Check that for all $i \in [q]$, the number of paths in B_i satisfies $m_i \geq (\ln n)^2/2$. If this does not hold then **stop**. The algorithm has failed.

Phase 7. Let $H = (V_H, E_H)$ be the graph defined by

$$V_H = \left\{(i, j) \mid i = 1, \dots, q; j = 1, \dots, m_i\right\}$$

and

$$E_H = \left\{\{(i, j), (i', j')\} \mid i \neq i' \text{ and } P_{i,j} \cap P_{i',j'} \neq \emptyset\right\}.$$

The ith *row* of H is the set of vertices $\{(i, j) \mid 1 \leq j \leq m_i\}$. A row represents the bundle of paths associated to a certain pair of endpoints, and a solution set corresponds to an independent set of size q that spans all the q rows of H.

Let Δ_H denote the maximum degree of a vertex in H. If there is an i such that $m_i \leq 8\Delta_H$ then **stop**, the algorithm has failed. (As shown in the analysis of this phase, the Local Lemma implies that the condition $m_i > 8\Delta_H$ is sufficient for the existence of at least one solution set.)

Optionally, for efficiency reasons, we can arbitrarily delete paths from each bundle until for every i we have $m_i = 8\Delta_H + 1$.

Phase 8. Let $H' = ([q], E_{H'})$ be the graph on q vertices defined by

$$E_{H'} = \left\{\{i, i'\} \mid \exists j, j' \text{ s.t. } P_{i,j} \cap P_{i',j'} \neq \emptyset\right\}.$$

(In other words H' contains an edge from i to i' if and only if any of the paths from $\tilde{a}_i$ to $\tilde{b}_i$ intersects any of the paths from $\tilde{a}_{i'}$ to $\tilde{b}_{i'}$. Clearly H' can be obtained from H by contracting each row of H to a single vertex.)

If any connected component of H' has size greater than $3 \ln n/(2 \ln \ln n)$ then **stop**. The algorithm has failed.

Phase 9. For each connected component J of H', find by exhaustive search an independent set in H of size $|J|$ that spans the rows of H corresponding to the vertices of J. (We checked in phases 6 and 7 that such a set exists, and we checked in phase 8 that the components of H' are sufficiently small to ensure that the exhaustive search takes only polynomial time.)

The union of independent sets thus found is independent and spans all the rows of H, and hence corresponds to a solution set.

The final path from a_i to b_i is the union of the paths from a_i to $\tilde{a}_i$, and from b_i to $\tilde{b}_i$ found in phase 4, and the path from $\tilde{a}_i$ to $\tilde{b}_i$ selected here.

End DisjPaths

Observation: Phases 6 and 7 are essential only for the proof and can be omitted while running the algorithm. In this case, the exhaustive search in phase 9 would still take only polynomial time, but with small probability (corresponding to failure in phase 6 or 7) it might not find a solution set. Nevertheless, the running time is likely to be shortened if phases 6 and 7 are run, since the search space is reduced.

4. Analysis of the algorithm.

4.1. Splitting expanders. In this subsection we present an algorithm that partitions the edge set of the input graph into two spanning expanders.

Algorithm Split

Input: An r-regular (α, β, γ)-expander graph $G = (V, E)$. For simplicity we assume that $r = 4s$, for an integer s.

Output: Two spanning β'-expanders $G_R = (V, E_R)$ and $G_B = (V, E_B)$ such that $E = E_R \cup E_B$ and $E_R \cap E_B = \emptyset$. (The constant β' is greater than 1 and will be exposed in the proof.)

1. Using an arbitrary Euler tour, orient the edges of G in such a way that each vertex has indegree and outdegree $2s$.

2. For each vertex v, randomly divide the edges from v into a red set and a blue set, each of size s. Set E_R (respectively, E_B) to be the set of red (respectively, blue) edges, un-oriented.

End Split

Clearly, our construction guarantees that (1) holds, since in each subgraph every vertex has degree at least s and at most $3s$, and each subgraph has exactly ns edges.

We now analyze the probability that **Split** will produce useful results. We start by defining two functions, H and ψ, on [0,1]:

$$H(\gamma) = ((1-\gamma)^{1-\gamma}\gamma^{\gamma})^{-1},$$

$$\psi(\epsilon) = (1-\epsilon)\ln(1-\epsilon) + \epsilon.$$

(Observe that $\psi(\epsilon) \geq \epsilon^2/2$.)

Let $\text{in}_R(S)$, $\text{out}_R(S)$ refer to $\text{in}(S)$ and $\text{out}(S)$ as applied to the graph G_R.

THEOREM 4.1. *Suppose that G is an (α, β, γ)-expander and let $0 < \epsilon < 1$ be such that*

$$\beta > \frac{2}{\psi(\epsilon)}\gamma^{-1}\ln H(\gamma). \tag{2}$$

For every set $S \subset V$, $|S| \leq |V|/2$, we have

$$\min\{\text{out}_R(S), \text{out}_B(S)\} \geq \min\{\alpha, (1-\epsilon)\beta/2\}\,|S|\,, \tag{3}$$

with probability $1 - o(1)$ as $n \to \infty$.

Proof. We obtain a lower bound for out_R. We consider two cases.

Case 1. $|S| \leq \gamma n$. By construction every vertex has degree at least s in G_R. Hence

$$\begin{aligned} s|S| &\leq 2\,\text{in}_R(S) + \text{out}_R(S) \\ &\leq 2\,\text{in}(S) + \text{out}_R(S). \end{aligned} \tag{4}$$

On the other hand, by the definition of G,

$$\begin{aligned} 4s|S| &= 2\,\text{in}(S) + \text{out}(S) \\ &\geq 2\,\text{in}(S) + (3s+\alpha)|S|. \end{aligned} \tag{5}$$

Inequalities (4) and (5) imply

$$\mathrm{out}_R(S) \geq \alpha|S|. \tag{6}$$

Case 2. $\gamma n \leq |S| \leq \frac{n}{2}$. Partition out($S$) so that two edges are in the same subset if in the Euler orientation they have the same start vertex.

Let there be m such sets, $A_1, \dots, A_m$, with $|A_i| = k_i \leq 2s$, and $\sum_{i=1}^m k_i = k$, where $k \geq \beta|S|$ by the definition of G. Let Z_i be the number of edges of A_i that are colored red. Clearly the Z_i's are independent. For any $t > 0$ and $\frac{k}{2} > u > 0$ we have

$$\begin{aligned}
&\mathbf{Pr}(Z_1 + \cdots + Z_m \leq k/2 - u)\\
&= \mathbf{Pr}\Big(\exp\big(-t(Z_1 + \cdots + Z_m - k/2 + u)\big) \geq 1\Big)\\
&\leq \mathbf{E}\Big(\exp\big(-t(Z_1 + \cdots + Z_m) - k/2 + u\big)\Big)\\
&= e^{t(k/2-u)} \prod_{i=1}^{m} \mathbf{E}(e^{-tZ_i}).
\end{aligned}$$

But

$$\begin{aligned}
\mathbf{E}(e^{-tZ_i}) &= \binom{2s}{s}^{-1} \sum_{j=0}^{k_i} \binom{k_i}{j}\binom{2s-k_i}{s-j} e^{-tj}\\
&\leq \left(1 + (e^{-t} - 1)\frac{k_i}{2s}\right)^s\\
&\leq \exp\big((e^{-t} - 1)k_i/2\big).
\end{aligned}$$

For a proof of the first inequality above see either Hoeffding [14, §6] or Chvátal [9]. (Note that although $Z_1, Z_2, \dots, Z_m$ are independent, a simple application of Theorem 2 of [14] will not suffice. This is because the Z_i have too large a range. The interested reader can check that one obtains a factor s^{-2} in the exponent of the probability bound.)

Hence

$$\begin{aligned}
&\mathbf{Pr}\big(Z_1 + \cdots + Z_m \leq k/2 - u\big)\\
&\quad\leq \exp\big(t(k/2 - u) + (k/2)(e^{-t} - 1)\big).
\end{aligned} \tag{7}$$

Putting $t = -\ln(1 - 2u/k)$ minimizes the right-hand side (RHS) of (7), which then becomes $\exp\big(-(k/2 - u)(\ln(1 - 2u/k)) - u\big)$. Hence if $u = \epsilon k/2$, then

$$\mathbf{Pr}\big(Z_1 + \ldots + Z_m \leq (1 - \epsilon)k/2\big) \leq e^{-k\psi(\epsilon)/2}$$

and consequently

$$\mathbf{Pr}\big(\mathrm{out}_R(S) \leq (1 - \epsilon)\beta|S|/2\big) \leq e^{-\beta|S|\psi(\epsilon)/2}.$$

Thus

$$\mathbf{Pr}\big(\exists|S| \geq \gamma n \text{ such that } \mathrm{out}_R(S) \leq (1 - \epsilon)\beta|S|/2\big) \leq \sum_{i \geq \gamma n} \binom{n}{i} e^{-\beta k\psi(\epsilon)/2}. \tag{8}$$

Now if $i = \theta n$, for $\theta \geq \gamma$ then $\binom{n}{i} = e^{o(n)}H(\theta)^n$, and the summand, u_i say, on the RHS of (8) is then

$$\exp\bigl(n(o(1) + \ln H(\theta) - \beta\theta\psi(\epsilon))/2\bigr).$$

Now

$$\theta^{-1}\ln H(\theta) = -\ln\theta + 1 - \frac{\theta}{2} - \frac{\theta^2}{6} - \frac{\theta^3}{12} - \cdots$$

clearly decreases with θ and so if β satisfies (2) then u_i is exponentially small. The result follows. □

COROLLARY 4.2. *Suppose that G is an* (α, β, γ)*-expander. Let* $0 < \epsilon_0 < 1$ *be the unique solution to*

$$(9)\qquad \frac{1-\epsilon}{\psi(\epsilon)} = \frac{\gamma}{\ln H(\gamma)}$$

and let

$$\beta_0 = \frac{2}{\psi(\epsilon_0)}\gamma^{-1}\ln H(\gamma).$$

If $\alpha > 1$ *and* $\beta > \beta_0$ *then both* G_R *and* G_B *are* β'*-expanders for some* $\beta' > 1$, *with probability* $1 - o(1)$.

Proof. The existence of ϵ_0 follows from the fact that the left-hand side of (9) decreases from ∞ to 0 as ϵ increases from 0 to 1. Plugging $\beta > \beta_0$ in (3), we get that $\beta' > 1$. □

It is fairly easy to apply this to the Ramanujan graphs of Lubotsky, Phillips, and Sarnak [17] and to random regular graphs. It follows from Lemma 2.3 of Alon and Chung [4] that

$$(10)\qquad |X| = \delta n \quad \text{implies } \operatorname{out}(X) \geq r(1-\lambda)(1-\delta)|X|,$$

where λ is the second largest eigenvalue of the transition probability matrix associated with the random walk on G. If G is one of the Ramanujan graphs then $\lambda = 2\sqrt{r-1}/r$ and if G is a large random r-regular graph then $\lambda \approx 2/\sqrt{r}$ (see Friedman, Kahn, and Szemerédi [13]). One can then show that in these cases $\min\{\operatorname{out}_R(S), \operatorname{out}_G(S)\} \geq (r/4 - o(1))\,|S|$ for $|S| \leq |V|/2$, as r grows. (For simplicity take $\gamma = \epsilon = r^{-1/3}$.)

The above ideas can be extended to arbitrary graphs. We need to be able to assert that (i) small sets of vertices, $|S| \leq \gamma n$, contain *few* edges; and that (ii) one can orient the edges so that every vertex has *large* outdegree. Given (ii) we can then randomly split the edges into two sets. It is known [11], [12] that the edge set of a graph can be oriented so that the out-degree of each vertex is at least k if and only if $|\mu(S)| \geq k|S|$ for all $S \subseteq V$ where $\mu(S) = \{e \in E : e \cap S \neq \emptyset\}$, and that this can be checked in polynomial time. We do not consider this generalization in this paper, however.

4.2. Analysis of the main algorithm. Let P denote the transition probability matrix of the random walk on G_B, and let $P^{(t)}_{v,w}$ denote the probability that the walk is at w at step t given that it started at v. Let λ be the second largest eigenvalue of P. (All eigenvalues of P are real.) It is known that

$$(11)\qquad P^{(t)}_{v,w} = \pi(w) + O(\lambda^t\sqrt{\pi(w)/\pi(v)}).$$

To ensure rapid convergence we need $\lambda \leq 1 - \epsilon$ for some *constant* $\epsilon > 0$. This holds for all expanders [1]. In particular if

$$\text{out}_B(S) \geq \beta'|S| \qquad \forall S \subseteq V, |S| \leq |V|/2, \tag{12}$$

for some *constant* $\beta' > 0$, Sinclair and Jerrum [22] show that (12) implies

$$\lambda \leq 1 - \frac{1}{2}\left(\frac{\beta'}{r}\right)^2.$$

We will now explicitly state our claims about the performance of our algorithm. As input, G is an n-vertex, bounded degree, r-regular (α, β, γ)-expander graph where $\alpha > 1$ and $\beta > \beta_0$, with β_0 as in Corollary 4.2.

Suppose that

$$\kappa > \max\{7, \kappa_1 \ln r, 2 + \kappa_3 \ln r\}, \tag{13}$$

$$\kappa_1 > \frac{4 + 2\kappa_3 \ln r}{\ln \lambda^{-1}} + \kappa_3, \tag{14}$$

$$\kappa_2, \kappa_3 > \frac{3}{\ln \lambda^{-1}}. \tag{15}$$

THEOREM 4.3. *Under the above assumptions with n sufficiently large, given any set of $q = n/(\log n)^{\kappa}$ disjoint pairs of vertices in G such that $\alpha > 1$ and $\beta > \beta_0$, with high probability our algorithm finds in $o(n^3)$ time, edge-disjoint paths connecting these q pairs.*

In §3 we pointed out for each phase the conditions under which it might fail. We now proceed to bound the associated failure probabilities.

Phase 1. The failure probability of this phase is $o(1)$ by Corollary 4.2. Also the time to carry out the construction is $O(n)$.

Phase 3. The $\kappa_1 \ln\ln n$ neighborhood of any vertex contains at most $\nu = r^{\kappa_1 \ln\ln n} = (\ln n)^{\kappa_1 \ln r}$ vertices. Using (1), the probability that r_i is rejected is thus never more than $3q\nu/2n$. Thus the probability that this phase fails is at most

$$\mathbf{Pr}(B(4q, 3q\nu/2n) \geq 2q)$$

and this is $o(1)$ if

$$\kappa_1 \ln r < \kappa, \tag{16}$$

since $q \leq n/(\ln n)^{\kappa}$. It is of course straightforward to carry out this selection in $o(n^2)$ time.

Phase 4. A straightforward application of the Max-Flow Min-Cut Theorem shows that success is certain provided that G_R is a β'-expander for some $\beta' > 1$. By Corollary 4.2 this happens with probability $1 - o(1)$. Furthermore it only takes $o(n^3)$ time to find the required flow since arc capacities are 1 for the arcs of the network.

Phase 5. The remainder of the proof relies heavily on the fact that the trajectories $W'_{i,j}$ constructed by our algorithm, have the same distribution (up to negligible factors) as m independent random trajectories of length $\tau = \kappa_2 \ln n$ from $\tilde{a}_i$, the difference being that we pick the endpoint of the trajectory using π instead of $P^{(\tau)}_{\tilde{a}_i,\cdot}$. Using (11), since

$$\kappa_2 > \frac{3}{\ln \lambda^{-1}} \tag{17}$$

we see that

$$|P^{(\tau)}_{v,w} - \pi(w)| = O(n^{-3})$$

for all v, w.

To allow us to view the trajectories $W'_{i,j}$, $W''_{i,j}$ as having *exactly* the same distribution as random trajectories we can imagine generating $W'_{i,j}$ as follows.

(a) Choose $x = x_{i,j}$ according to the distribution $P^{(\tau)}_{\tilde{a}_i,\cdot}$.
(b) Choose a random trajectory $W'_{i,j}$ from $\tilde{a}_i$ to x.
(c) If $\theta(x) = P^{(\tau)}_{\tilde{a}_i,x} - \pi(x) > 0$ then with probability $\theta(x)/P^{(\tau)}_{\tilde{a}_i,x}$ do
  1. discard $W'_{i,j}$;
  2. choose $y \in \Omega^- = \{v : \theta(v) < 0\}$ with probability $\theta(y)/\theta(\Omega^-)$;
  3. choose a new random trajectory $W'_{i,j}$ from $\tilde{a}_i$ to y.

It is not hard to see that the endpoint of $W'_{i,j}$ other than $\tilde{a}_i$ is now chosen according to the distribution π. Furthermore, as long as (1)–(3) above are not executed, we can view $W'_{i,j}$ as a random walk of length τ from $\tilde{a}_i$. But

$$\begin{aligned}&\mathbf{Pr}((1) - (3) \text{ occur during the algorithm})\\ &= O(qm \max \theta(x)) = O((\ln n)^{2-\kappa}/n) = o(1).\end{aligned}$$

This justifies viewing the $W'_{i,j}$, $W''_{i,j}$ as unbiased random walks.

The next question is how, given $x_{i,j}$, do we compute a random trajectory of length τ from $\tilde{a}_i$ to $x_{i,j}$? This is not difficult.

To simplify notation, suppose we want to compute a random trajectory $W = [u_0 = u, u_1, \ldots, u_t = v]$ of length t from a vertex u to a vertex v. If w is a neighbor of v then

$$\mathbf{Pr}(u_{t-1} = w | u_t = v) = \frac{P^{(t-1)}_{u,w} P_{w,v}}{P^{(t)}_{u,v}}. \tag{18}$$

Thus our algorithm to generate W is to choose w according to (18) and then choose a random trajectory of length $t - 1$ from u to w. To compute $P^{(t)}$ we need only compute powers of P. Because G has bounded degree we can compute P^k from P^{k-1} in $O(n^2)$ time. Thus the total time to compute all the trajectories is $O(\tau n^2) = o(n^3)$, for $\kappa > 7$.

Phase 6. We prove several intermediate propositions. Our aim is to show that relatively few paths get deleted.

PROPOSITION 4.4. *Assume that*

$$\kappa_1 \geq \frac{4 + 2\kappa_3 \ln r}{\ln \lambda^{-1} + \kappa_3}. \tag{19}$$

Then with probability $1 - o(1)$ *the number of paths deleted due to condition* (a) *is* $O(\ln n)$ *simultaneously for each* $i \in [q]$.

Proof. Recall that $M_{i,j} = \{w'_{i,j,t}, w''_{i,j,t} : t \geq (\kappa_1 - \kappa_3) \ln \ln n\}$, and a path $P_{i,j}$ is deleted due to condition (a) if $\operatorname{dist}(M_{i,j}, \bigcup_{k<j} M_{i,k}) \leq 2\kappa_3 \ln \ln n$.

For $t \geq (\kappa_1 - \kappa_3) \ln \ln n$ the probability that $w'_{i,j,t} = v$ is (by (11)) $O(\lambda^t + 1/n) = O((\ln n)^{-(\kappa_1-\kappa_3)\ln \lambda^{-1}})$ for any vertex v. Also the $2\kappa_3 \ln \ln n$ neighborhood of $\bigcup_{k<j} M_{i,k}$ is of size $O((\ln n)^{3+2\kappa_3 \ln r})$ and so the probability that $W'_{i,j}$ or $W''_{i,j}$ wanders into this neighborhood after $(\kappa_1 - \kappa_3) \ln \ln n$ steps, is only

$$O((\ln n)^{3+2\kappa_3 \ln r-(\kappa_1-\kappa_3)\ln \lambda^{-1}}) = O(1/\ln n),$$

given (19). Thus the number of paths deleted from bundle i is dominated by a binomial random variable $B(N, p)$ with $Np = O(\ln n)$.

The inequality (see, e.g., [7, Thm. I.7])

$$\mathbf{Pr}(B(N, p) \geq aNp) \leq \left(\frac{e}{a}\right)^{aNp} \tag{20}$$

is, for sufficiently large a, enough to verify the proposition. (Take $a = c \ln n/(Np)$ for a sufficiently large c.) □

PROPOSITION 4.5. *Assume that*

$$\kappa \geq 2 + \kappa_3 \ln r. \tag{21}$$

Let

$$N_i = \{v \in R - \{\tilde{a}_i, \tilde{b}_i\} : \text{dist}(v, B_i) \leq \kappa_3 \ln \ln n\}.$$

Then $|N_i| = O(\ln n)$ *simultaneously for each* $i \in [q]$, *with probability* $1 - o(1)$.

Proof. The size of the $\kappa_3 \ln \ln n$ neighborhood of all the bundles B_i together is $O((\ln n)^{3+\kappa_3 \ln r})$. The number of vertices in R chosen in this neighborhood is a binomial with mean $O(\ln n)$, given (21). The result follows again by using (20). □

We can now bound the number of paths deleted from each bundle in phase 6 due to condition (b). Recall that the vertices of $Q \setminus \{\tilde{a}_i\}$ are at least $\kappa_1 \ln \ln n$ away from $\tilde{a}_i$. Hence, if two paths in a bundle are simultaneously closer than $\kappa_3 \ln \ln n$ to the endpoint of another bundle, then one of them is deleted by condition (a). Thus any $v \in N_i \cap Q$ can lead to the deletion of a single path via condition (b), so almost surely only a total $O(\ln n)$ paths are deleted from each bundle.

Phase 7. We start by proving the following proposition.

PROPOSITION 4.6. *The maximum degree in the graph* H (*the incidence graph of the paths*) *satisfies* $\Delta_H = O((\ln n)^2/\ln \ln n)$, *almost surely.*

Proof. We will show below in the analysis of phase 8 that with probability $1 - o(1)$ the graph H' has maximum component size $O(\ln n/\ln \ln n)$ and so it suffices to prove that with probability $1 - o(1)$ for every i, j, k, the trajectory $W'_{i,j}$ meets only $O(\ln n)$ trajectories in the bundle B_k.

Now fix i, j, k. The pruning done in phase 6 allows us to assume now that $\text{dist}(W'_{i,j}, \{\tilde{a}_k, \tilde{b}_k\})$ is at least $\kappa_3 \ln \ln n$. Consider a trajectory $W'_{k,l}$. The probability that $W'_{k,l}$ meets $W'_{i,j}$ is by (11) of order $O((\ln n)^{2-\kappa_3 \ln \lambda^{-1}}) = O(1/(\ln n))$ provided that

$$\kappa_3 \geq \frac{3}{\ln \lambda^{-1}}. \tag{22}$$

Treating the construction of each $W'_{k,l}$ as an independent trial we see that the expected number of trials in which $W'_{i,j} \cap W'_{k,l} \neq \emptyset$ is $O(\ln n)$. We can now use (20). □

We now show that if we reach the start of phase 7 and $m_i > 8\Delta_H$ for each i then we can be sure that there is a set of disjoint paths contained in our bundles. We use the following lemma [10], [23].

LOVÁSZ LOCAL LEMMA. *Let* $A_1, \ldots, A_N$ *be events with dependency graph* G_A. *Let* $\deg(i)$ *be the degree of* A_i *in* G_A. *If*

$$\begin{aligned} \mathbf{Pr}(A_i) &\leq p \quad \forall \text{i}, \\ \deg(i) &\leq d \quad \forall \text{i}, \\ 4pd &< 1, \end{aligned}$$

then

$$\mathbf{Pr}(\wedge \bar{A_i}) > 0.$$

Consider the experiment in which a random vertex is chosen from each row of H. The events A_i (the "bad" events) are defined by the choice of two vertices joined by an edge. The maximum degree in the dependency graph for the Lovász Local Lemma is $2m\Delta_H$ and each bad event has probability at most $4/m^2$. The Local Lemma now proves easily that our independent set exists, since $m = (\ln n)^2$ and $\Delta_H = O((\ln n)^2/\ln\ln n)$.

(Alon has shown in [2] the existence of such spanning independent sets in a similar setting using the same technique. Later he proved in [3] that for m sufficiently large relative to the dependency degree there is a proper m-coloring of the graph so that each row uses m colors. Furthermore, Beck has shown that this coloring can be found in polynomial time [6]. However, for our purposes it is necessary to show that the component size does not exceed $O(\log n/\log\log n)$ (see below), hence these more sophisticated algorithms seem unnecessary.)

Phase 8. Pair the vertices in $R \setminus Q$ arbitrarily. Connect each of the q new pairs by m random walks as is phase 5. Define a supergraph $H'' \supset H'$ obtained from H' by adding q additional vertices corresponding to bundles of paths connecting vertices of $R \setminus Q$, and by adding extra edges in an obvious way (we ignore the pruning of phase 6).

Let $\pi(r)$ be the other end of a bundle that has endpoint r. Let $\tilde{W}_r$ be the set of m random walks of length τ starting at vertex $r \in R$. We claim that if r is fixed and r' is chosen uniformly at random from $R \setminus \{r, \pi(r)\}$, then there exists a constant κ_4 such that the probability that $\tilde{W}_r$ and $\tilde{W}_{r'}$ intersect is bounded by

$$\mathbf{Pr}(\tilde{W}_r \cap \tilde{W}_{r'} \neq \emptyset) \leq \frac{\kappa_4(\ln n)^6}{4n}. \tag{23}$$

To see this, consider a random walk from r' of length τ and assume for the moment that r' is chosen uniformly at random in R. Since r' is thus chosen from the steady state of the random walk, the expected number of vertices of $\tilde{W}_r$ visited by this random walk is $O(m\tau^2/n)$. Summing over all walks in $\tilde{W}_{r'}$ we obtain $O(m^2\tau^2/n)$ as the expected number of visits to $\tilde{W}_r$. The probability of at least one visit is bounded by this expectation. Now if r' is chosen uniformly at random only within $R \setminus \{r, \pi(r)\}$, the probability of intersection increases at most by a factor of $(1 - 2/|R|)^{-1} = 1 + o(1)$. Thus we have (23).

Each bundle B_i is composed of two sets of m random walks. Hence, for i fixed, and i' chosen uniformly at random in $\{1, \ldots, 2q\} \setminus \{i\}$ we have

$$\mathbf{Pr}(\{B_i, B_{i'}\} \text{ is an edge of } H'') \leq \kappa_4 \frac{(\ln n)^6}{n}. \tag{24}$$

Furthermore, if σ is a random permutation of $[2q]$,

$$\mathbf{Pr}(H'' \text{ contains a component of size } \geq k)$$

$$\leq \sum_{\substack{S\subseteq[2q]\\ |S|=k}} \sum_{T\in\Omega_{\sigma(S)}} \mathbf{Pr}(\mathcal{E}_T) = \binom{2q}{k} \sum_{T\in\Omega_{\sigma([k])}} \mathbf{Pr}(\mathcal{E}_T),$$

where Ω_A denotes the set of trees with vertex set A, and $\mathcal{E}_T$ denotes the event that H'' contains a tree isomorphic to T, under the isomorphism $i \leftrightarrow B_i$. The inequality is immediate because any component of size $\geq k$ must contain a tree of size k, and the equality follows from symmetry.

We can restrict our attention to the range where $k/q < \frac{1}{2}$. We claim that $\mathbf{Pr}(\mathcal{E}_T) \leq (2\kappa_4(\ln n)^6/n)^{k-1}$. Indeed consider the edges of T in a breadth-first search order from some arbitrary root; assume that we have already explored $l-1$ edges, and thus l vertices numbered without loss of generality $1, \ldots, l$. The probability that the lth edge exists is given by the probability that for a certain fixed $i \in \{1, \ldots, l\}$ the bundle B_i intersects $B'_{i'}$ where i' is chosen uniformly at random (via σ) in $\{l+1, \ldots, 2q\}$. This probability can be proven via essentially the same argument used to derive (23) and (24) to be less than $(1-(2l)/(2q))^{-1}\kappa_4(\ln n)^6/n$, independently of the existence of previous edges.

Now, since $|\Omega_{\sigma([k])}| = k^{k-2}$ we obtain that

$$\begin{aligned}
&\mathbf{Pr}(H'' \text{ contains a component of size } \geq k)\\
&= O\left(\left(\frac{2qe}{k}\right)^k \left(\frac{2\kappa_4(\ln n)^6}{n}\right)^{k-1} k^{k-2}\right)\\
&= O\left(\left(\frac{n}{(\ln n)^6}\left(\frac{4e\kappa_4 q(\ln n)^6}{n}\right)^k\right)\right)\\
&= O\left(\frac{n}{(\ln n)^6}\left(\frac{4e\kappa_4}{(\ln n)^{\kappa-6}}\right)^k\right) = o(1)
\end{aligned}$$

for $\kappa > 7$ and $k \geq k_0 = \ln n/(\ln\ln n)$.

Phase 9. The execution time of Phase 9, given that there are no large components in H'', is bounded by

$$\frac{n}{(\ln n)^\kappa}((\ln n)^2)^{\ln n/\ln\ln n} = o(n^3).$$

5. Random $\mathcal{NC}$ algorithms.

In this section we show that our construction can be done in random $\mathcal{NC}$. To convert **DisjPaths** to a random $\mathcal{NC}$ algorithm we need to modify phases 2 and 3 of the algorithm. We replace them by the following two phases:

Phase 2*. Each vertex $v \in V$ is included in R with probability $8q\pi_v$ independent of the other vertices.

Phase 3*. A vertex $u \in R$ is in Q if no vertex in its $\kappa_1 \ln\ln n$ neighborhood is in R.

We now consider each phase in turn.

Phase 1. The algorithm **Split** is in $\mathcal{NC}$ since computing an Euler Path is in $\mathcal{NC}$ [5].

Phases 2* and 3*. With probability $1-o(1)$, R has at least $4q$ vertices. The probability that a vertex in R has another vertex in R in its $\kappa_1 \ln\ln n$ neighborhood is smaller than $\frac{1}{2}$, thus with probability $1-o(1)$, Q has at least $2q$ vertices. The fact that Q might have more than $2q$ vertices does not matter since the flow algorithm gives an integer solution, and only $2q$ vertices in Q will participate in the flow.

Phase 4. Flow with unit capacities is in Random $\mathcal{NC}$ [15], [18].

Phases 5, 6, and 7. By attaching one processor to each of the $q(\ln n)^2$ paths used in the algorithm, all these phases can be computed in $O(\ln n)$ time.

Phase 8. Computing connected components is in $\mathcal{NC}$.

Phase 9. Observe that there are no more than $n/(\ln n)^{\kappa}$ components, and with high probability there are no more than

$$((\ln n)^2)^{\ln n/\ln\ln n} = n^2$$

choices of paths for each component. Given a possible choice, it can be checked by one processor in $O(\ln^2 n)$ steps. Thus, phase 9 can be computed by $o(n^3)$ processors in $O(\ln^2 n)$ parallel steps.

REFERENCES

[1] N. Alon, *Eigenvalues and expanders*, Combinatorica, 6 (1986), pp. 83–96.

[2] ———, *The linear arboricity of graphs*, Israel J. Math., 62 (1988), pp. 311–325.

[3] ———, *The strong chromatic number of a graph*, Random Structures Algorithms, 3 (1992), pp. 1–8.

[4] N. Alon and F. R. K. Chung, *Explicit construction of linear sized tolerant networks*, Discrete Math., 72 (1989), pp. 15–19.

[5] B. Awerbuch, A. Israeli, and Y. Shiloach, *Finding Euler circuits in logarithmic parallel time*, in Advances in Computing Research 4: Parallel and Distributed Algorithms, F. P. Preparata, ed., JAI Press, Greenwich, CT, 1987, pp. 69–78.

[6] J. Beck, *An algorithmic approach to the Lovász Local Lemma* I, Random Structures Algorithms, 2 (1991), pp. 343–365.

[7] B. Bollobás, *Random Graphs*, Academic Press, New York, 1985.

[8] A. Z. Broder, A. M. Frieze, and E. Upfal, *Existence and construction of edge-disjoint paths on expander graphs*, in Proceedings of 24th Annual ACM Symposium on Theory of Computing, Victoria, BC, Canada, ACM Press, New York, 1992, pp. 140–149.

[9] V. Chvátal, *Probabilistic methods in graph theory*, Ann. Oper. Res., 1 (1984), pp. 171–182.

[10] P. Erdös and L. Lovász, *Problems and results on 3-chromatic hypergraphs and some related questions*, Colloq. Math. Soc. János Bolyai, 11 (1975), pp. 609–627.

[11] T. I. Fenner and A. M. Frieze, *On the connectivity of random m-orientable graphs and digraphs*, Combinatorica, 2 (1982), pp. 347–359.

[12] A. Frank and A. Gyarfas, *How to orient the edges of a graph*, Colloq. Math. Soc. János Bolyai, 18 (1978), pp. 353–364.

[13] J. Friedman, J. Kahn, and E. Szemerédi, *On the second eigenvalue in random regular graphs*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle, WA, ACM Press, New York, 1989, pp. 587–598.

[14] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, J. Amer. Statist. Assoc., 58 (1963), pp. 13–30.

[15] R. M. Karp, E. Upfal, and A. Wigderson, *Constructing a perfect matching is in random NC*, Combinatorica, 6 (1986), pp. 35–48.

[16] T. Leighton and B. Maggs, *Expanders might be practical: Fast algorithms for routing around faults in multibutterflies*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, IEEE Computer Society Press, New York, 1990, pp. 384–389.

[17] A. Lubotsky, R. Phillips, and P. Sarnak, *Ramanujan graphs*, Combinatorica, 8 (1988), pp. 261–277.

[18] K. Mulmuley, V. Vazirani, and V. Vazirani, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[19] D. Peleg and E. Upfal, *Constructing disjoint paths on expander graphs*, Combinatorica, 9 (1989), pp. 289–313.

[20] ———, *The token distribution problem*, SIAM J. Comput., 18 (1989), pp. 229–243.

[21] N. Robertson and P. D. Seymour, *Graph minors*-XIII*: The disjoint paths problem*, J. Combinatorial Theory, Ser. B, to appear.

[22] A. Sinclair and M. Jerrum, *Approximate counting, uniform generation, and rapidly mixing Markov chains*, Inform. and Comput., 82 (1989), pp. 93–133.

[23] J. Spencer, *Ten Lectures on the Probabilistic Method*, SIAM, Philadelphia, 1987.

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

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 990–1000, October 1994

SEPARATING DISTRIBUTION-FREE AND MISTAKE-BOUND LEARNING MODELS OVER THE BOOLEAN DOMAIN*

AVRIM L. BLUM†

Abstract. Two of the most commonly used models in computational learning theory are the distribution-free model in which examples are chosen from a fixed but arbitrary distribution, and the absolute mistake-bound model in which examples are presented in an arbitrary order. Over the Boolean domain $\{0, 1\}^n$, it is known that if the learner is allowed unlimited computational resources then any concept class learnable in one model is also learnable in the other. In addition, any polynomial-time learning algorithm for a concept class in the mistake-bound model can be transformed into one that learns the class in the distribution-free model.

This paper shows that if one-way functions exist, then the mistake-bound model is strictly harder than the distribution-free model for polynomial-time learning. Specifically, given a one-way function, it is shown how to create a concept class over $\{0, 1\}^n$ that is learnable in polynomial time in the distribution-free model, but not in the absolute mistake-bound model. In addition, the concept class remains hard to learn in the mistake-bound model even if the learner is allowed a polynomial number of membership queries.

The concepts considered are based upon the Goldreich, Goldwasser, and Micali random function construction [Goldreich, Goldwasser, and Micali, *Journal ACM*, 33 (1986), pp. 792–807] and involve creating the following new cryptographic object: an exponentially long sequence of strings $\sigma_1, \sigma_2, \dots, \sigma_r$ over $\{0, 1\}^n$ that is hard to compute in one direction (given σ_i one cannot compute σ_j for $j < i$) but is easy to compute *and even make random-access jumps* in the other direction (given σ_i and $j > i$ one *can* compute σ_j, even if j is exponentially larger than i). Similar sequences considered previously [Blum, Blum, and Shub, *SIAM J. Comput.*, 15 (1986), pp. 364–383], [Blum and Micali, *SIAM J. Comput.*, 13 (1984), pp. 850–863] did not allow random-access jumps forward without knowledge of a seed allowing one to compute backwards as well.

Key words. machine learning theory, learning models, one-way functions

AMS subject classifications. 68Q25, 68T05, 94A60

1. Introduction.

Two of the most popular theoretical models for learning from examples are the distribution-free model, also known as the *probably-approximately-correct* (PAC) or "Valiant-style" learning model [17], [25], and the absolute mistake-bound model [21]. In both models, the goal of a learner is to approximately infer some unknown target concept from positive and negative examples of that concept. In the distribution-free model, an adversary chooses a distribution over the labeled examples from which the learner may sample. The learner, from a polynomial (in relevant parameters discussed in §2) number of samples, must produce a hypothesis that agrees with the target concept over most of the distribution. In the mistake-bound model, the adversary instead actually chooses the order in which examples appear. Here the learner sees unlabeled examples, and after each one must predict whether it is positive or negative before being told its true classification. The job of the learner in the mistake-bound model is to make a polynomially bounded number of mistakes.

In this paper, we will consider the "representation-independent" versions of the distribution-free and mistake-bound models, also known as the "prediction" model [12], [13], in which the learner's hypotheses need not be the same form as the target concept. The mistake-bound model is equivalent to Angluin's equivalence query model [3], [21] when query hypotheses are similarly not restricted.

It is known that any algorithm for learning a concept class in the mistake-bound model can be converted to one that will learn the class in the distribution-free model [3], [18]. If computational considerations are ignored, then over the Boolean domain $\{0, 1\}^n$ the converse holds as well. Any concept class over $\{0, 1\}^n$ learnable with polynomial sample size (and

*Received by the editors July 20, 1992; accepted for publication (in revised form) July 4, 1993.

†School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3891 (avrim@cs.cmu.edu). This work was done while the author was at the MIT Lab for Computer Science and supported by an NSF Graduate Fellowship, NSF grant CCR-8914428 and the Siemens Corporation.

not necessarily polynomial time) in the distribution-free model can also be learned with a polynomial mistake bound (in not necessarily polynomial time) in the mistake-bound model [3], [21]. At the heart of the proof for this last direction is the use in the mistake-bound model of the "halving algorithm." This algorithm enumerates all concepts in the class, predicts according to majority vote, and when a mistake is made throws out the concepts that predicted incorrectly. The equivalence of learnability in the *non*computational setting has led researchers to wonder (e.g., [16]) whether the models remain equivalent when computation *is* limited—especially in light of recent work showing that the distribution-free model is equivalent to a seemingly much easier form of learning known as "weak-learning" [24].

In this paper we show that if we consider only algorithms that run in polynomial time, then if one-way functions exist, the distribution-free and mistake-bound models over $\{0, 1\}^n$ are *not* equivalent. Specifically, we create a concept class starting from a pseudorandom bit generator [6], [7], [26], which is learnable in polynomial time in the distribution-free model but not in the mistake-bound model. The existence of pseudorandom bit generators has been proven equivalent to the existence of one-way functions [20], [15], [11]. The concept class we create remains hard to learn in the mistake-bound model even if the learning algorithm is allowed membership queries—the ability to ask whether examples of its own choosing belong to the target concept. So, for instance, the position of this concept class contrasts with that of deterministic finite automata (DFAs) which *are* learnable in a mistake-bound model with membership queries [2] but not in the distribution-free model [19].

The concept class created is based upon the Goldreich, Goldwasser, and Micali random function construction [9] and involves creating the following new cryptographic object: an exponentially long sequence of strings $\sigma_1, \sigma_2, \ldots, \sigma_r$ over $\{0, 1\}^n$ that is hard to compute in one direction (given σ_i one cannot compute σ_j for $j < i$) but is easy to compute *and even make random-access jumps* in the other direction (given σ_i and $j > i$ one *can* compute σ_j, even if j is exponentially larger than i). This sequence is "stronger" than those considered previously [6], [7] in the sense that for those sequences, the only known way to compute in the forward direction (without knowing the seed that allows one to compute in the other direction as well) is to compute the strings sequentially in order.

The sequence of strings is useful for separating the learning models for roughly the following reason. Attached to each σ_i is a classification that can only be computed given σ_j for $j < i$. So, an adversary can present the strings in the reverse order $\sigma_r, \sigma_{r-1}, \ldots$ and cause the learner to make a mistake nearly half the time. However, for any distribution over the strings, if a learner collects m samples, we expect that about $m/(m + 1)$ of the distribution falls on the "easy side" (the set of σ_j for $j > i$) of the string σ_i of least index seen. The "random-access forward" property of the sequence then allows the distribution-free learner to use σ_i as a predictor for any σ_j for $j > i$. Notice that without the random-access property, then even on a uniform distribution the learner might still fail because the examples seen would likely all be exponentially far away from each other.

An earlier version of this paper appears in [5].

2. Notation, definitions, and background. An *example* x is an element of $\{0, 1\}^n$ and a concept is a set of examples (subset of $\{0, 1\}^n$). We will also identify concepts with their indicator functions, defining $c(x) = 1$ if $x \in c$ and $c(x) = 0$ otherwise. For a given target concept c, a *labeled example* of c is a pair $\langle x, c(x) \rangle$ where x is an example and $c(x)$ is its *classification*. An example x is a *positive example* if $c(x) = 1$; otherwise it is a negative example. A *concept class* is a collection of concepts, together with a (sometimes implicit) representation for each concept.[1] For instance, the class of DNF formulas consists of concepts

[1] To be rigorous, we should define a concept class C to be a family $\{C_n\}$ (one for each n) where C_n is defined over $\{0, 1\}^n$. In order to avoid overly cumbersome notation, we shall assume this type of indexing is done implicitly.

given in a disjunctive normal-form representation. If c is a concept in a given concept class C, we use $|c|$ to denote the size of the (smallest) representation of c under C. In both the distribution-free and mistake-bound models, the object of a learning algorithm for a concept class C is to approximately infer an unknown target concept $c \in C$ from examples. The models differ in how examples are chosen and how successful learning is defined.

In the distribution-free model, there is some distribution $\mathcal{D}$ over the set of labeled examples of the target concept. The learning algorithm is allowed to sample from $\mathcal{D}$ and based on this information and knowledge of the class C to which the target concept belongs, must produce a hypothesis h that approximates the target concept. In particular, if c is the target concept, we say a hypothesis h has *error* ε if on pair $\langle x, c(x)\rangle$ chosen from $\mathcal{D}$, the probability that $h(x)$ does not equal $c(x)$ is ε. We will allow the hypothesis produced by the algorithm to not necessarily belong to concept class C (this version is sometimes termed "polynomial predictability" [12], [13]). More formally, an Algorithm A *learns in the distribution-free model* a concept class C over $\{0, 1\}^n$ if for some polynomial P, for all target concepts $c \in C$, distributions $\mathcal{D}$, and error parameters ε and δ: in time at most $P(n, \frac{1}{\varepsilon}, \frac{1}{\delta}, |c|)$ the algorithm with probability at least $1 - \delta$ finds a polynomial-time evaluatable hypothesis h (not necessarily from C) with error at most ε. See Haussler, Kearns, Littlestone, and Warmuth [12] for various equivalent formulations of the distribution-free model.

In the mistake-bound model, instead of a distribution, we imagine there is some adversary presenting the examples in any order it wishes. Learning is done on-line in a sequence of *stages*. In each stage the adversary presents an unlabeled example to the learner, the learner suggests a labeling, and then the learner is told whether or not that labeling was correct. The object of the learning algorithm is to make at most a polynomial number of mistakes. Algorithm A *learns* a concept class C *in the mistake-bound model* if for some polynomial P, for all target concepts $c \in C$ and all orderings of the examples, the algorithm makes at most $P(n, |c|)$ mistakes using polynomial time in each stage [21].

A *membership query* is a query in which the learner selects an unlabeled example of its own choosing and asks for (and is told) that example's classification. We may incorporate membership queries into the mistake-bound model by allowing the learner at each stage to choose either to make a membership query or else to receive an example from the adversary. We will say that Algorithm A *learns* a concept class C *in the mistake-bound model with membership queries* if for some polynomial P, for all target concepts $c \in C$, the algorithm makes at most $P(n, |c|)$ mistakes and membership queries using polynomial time in each stage.

Note that in all models we require a learning algorithm to run in polynomial time. In some of the literature, this is called "polynomial learnability."

We now review some useful notation and definitions from the cryptographic literature. If S is a set we will use the notation $s \in_R S$ to mean that s is chosen uniformly at random from S. For convenience, if A is a probabilistic polynomial time (PPT) algorithm and g is some function, we will use $\mathcal{P}_d(A, g(s))$ to mean $\mathbf{Pr}\left[A(g(s)) = 1 \mid s \in_R \{0, 1\}^d\right]$. If $g(s) = s$ we will just write $\mathcal{P}_d(A, s)$. In order to simplify the statements of some of the theorems, in this paper we will use the term "algorithm" to include circuit families (nonuniform algorithms).

Informally, a cryptographically strong pseudorandom bit generator, or CSB generator to use the notation of GGM, is a deterministic polynomial-time algorithm that given a randomly chosen input produces a longer pseudorandom output. More formally we have the following.

DEFINITION 1. *A deterministic polynomial-time program G is a* CSB generator with stretch *t if on input $s \in \{0, 1\}^k$ it produces a tk-bit output, and for all probabilistic polynomial-time algorithms A, for all polynomials Q, for all sufficiently large k:*

$$|\mathcal{P}_k(A, G(s)) - \mathcal{P}_{tk}(A, s)| < \frac{1}{Q(k)}.$$

That is, no polynomial-time algorithm can distinguish with a $1/\text{poly}(k)$ probability between a string chosen randomly from $\{0, 1\}^{tk}$ and the output of G on a string chosen randomly from $\{0, 1\}^k$. For this paper, we will just need a CSB generator G with stretch 2.

For a CSB generator G that on input $s \in \{0, 1\}^k$ produces a $2k$-bit output $G(s) = b_1, \ldots, b_{2k}$, let us define the following notation.

(i) Let $G_0(s)$ be the leftmost k bits $b_1, \ldots, b_k$ of $G(s)$, and let $G_1(s)$ be the rightmost k bits $b_{k+1}, \ldots, b_{2k}$.

(ii) For a d-bit string $i = i_1 \ldots i_d \in \{0, 1\}^d$, let

$$G_{i_1 \cdots i_d}(s) = G_{i_d}(G_{i_{d-1}}(\ldots G_{i_2}(G_{i_1}(s)) \ldots)).$$

Note that this is well defined because $G_0(s)$ and $G_1(s)$ are both k-bit strings.

(iii) Let $G'_0(s) = G_1(s)$ and let $G'_1(s) = \lambda$ (i.e., the empty string). Think of G' as a rightward-shift of G.

If x and y are strings, we will use $x \circ y$ to denote the concatenation xy. Also, if an example created by a concatenation of strings has length less than n, then we will implicitly pad with zeros, say, on the right. Finally, let $\text{LSB}[x]$ denote the rightmost bit of string x.

We now review the GGM random function construction. Let G be a CSB generator that on input $s \in \{0, 1\}^k$ produces a $2k$-bit output. Goldreich, Goldwasser, and Micali define the function $f_s : \{0, 1\}^k \to \{0, 1\}^k$ on input $i = i_1 \ldots i_k$ to be

$$f_s(i) = G_{i_1 \cdots i_k}(s) = G_{i_k}(G_{i_{k-1}}(\ldots G_{i_1}(s) \ldots)).$$

The function f_s can be viewed as a complete binary tree of depth k. The root is the seed s, and on input i, the output $f_s(i)$ is the ith leaf in the tree if i is viewed as a binary number between 0 and 2^k. Though this fact will not be needed for our purposes, it is proved that the collection $F_k = \{f_s\}_{s \in \{0,1\}^k}$ is a "polyrandom" collection of functions over $\{0, 1\}^k$. Essentially, this means that no polynomial-time algorithm can distinguish between a function chosen randomly from F_k and a function chosen randomly from the class of all functions from $\{0, 1\}^k$ to $\{0, 1\}^k$.

We will use the basic form of the GGM construction as a starting point to create a concept class that separates the distribution-free and mistake-bound learning models.

3. A concept class to separate the learning models. We now create a concept class over $\{0, 1\}^n$ that is learnable in the distribution-free model but not in the mistake-bound model if one-way functions exist. We first give the formal definition of the class, and follow this by a more informal explanation and example. Given a CSB generator G, we will fix k, the size of seed s, to equal $\lfloor \sqrt{n} \rfloor$.

DEFINITION 2. *Let $\mathcal{C}_G = \{c_s\}_{s \in \{0,1\}^k}$ where the concepts c_s are defined as follows*:

$$c_s = \{x_s^i : i \in \{0, 1\}^k \text{ and } \text{LSB}[G_{i_1 \cdots i_k}(s)] = 1\},$$

where for each $i = i_1 \cdots i_k$,

$$x_s^i = i \circ G'_{i_1}(s) \circ G'_{i_2}(G_{i_1}(s)) \circ G'_{i_3}(G_{i_1 i_2}(s)) \circ \ldots \circ G'_{i_k}(G_{i_1 \cdots i_{k-1}}(s)).$$

Now the informal description. There is one concept c_s for each $s \in \{0, 1\}^k$. For each i such that $\text{LSB}[f_s(i)] = 1$, there is an associated positive example x_s^i for c_s, so there are about $2^{\lfloor \sqrt{n} \rfloor - 1}$ positive examples for c_s. We can think of the labeled example $\langle x_s^i, c_s(x_s^i) \rangle$ as the pair $(i, \text{LSB}[f_s(i)])$ together with the following extra information: for each bit i_j equal to zero (running left to right along i), we include the right half of $G(G_{i_1 \cdots i_{j-1}}(s))$. For example, if $k = 4$ and $i = 0101$, then we would have (see Fig. 1)

$$\begin{aligned} x_s^i &= 0101 \circ G_1(s) \circ \lambda \circ G_1(G_1(G_0(s))) \circ \lambda, \\ &= 0101 \circ G_1(s) \circ G_{011}(s), \\ c_s(x_s^i) &= \mathrm{LSB}[G_{0101}(s)]. \end{aligned}$$

We will think of the index i as both a bit string and a binary number between 0 and 2^k.

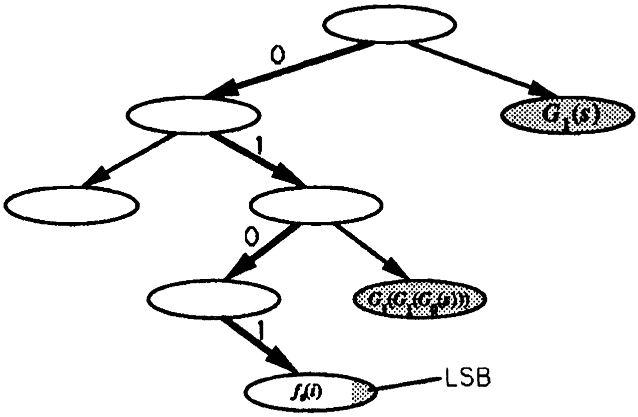


FIG. 1. *Darkened regions are information given with* $\langle x_s^i, c_s(x_s^i)\rangle$ *for* $i = 0101$.

Notice that negative examples of c_s come in two forms: some are examples x_s^i for which $\mathrm{LSB}[f_s(i)] = 0$ and some are just bit strings from $\{0, 1\}^n$ that are not of the form x_s^i for any i. We will call the examples x_s^i the "good examples" of c_s; so the positive examples of c_s are the good examples x_s^i such that $\mathrm{LSB}[f_s(i)] = 1$. The examples x_s^i will be shown to have the property of the strings σ_i mentioned in the introduction.

For convenience, let us define the following additional notation.

(i) Let z_s^i be the (correctly) labeled good example $\langle x_s^i, c_s(x_s^i)\rangle$.

(ii) Let $\bar{z}_s^i$ be the incorrectly labeled good example $\langle x_s^i, 1 - c_s(x_s^i)\rangle$.

(iii) For $i_1 \ldots i_d \in \{0, 1\}^d$, let:

$$G^{i_1 \cdots i_d}(s) = G'_{i_1}(s) \circ G'_{i_2}(G_{i_1}(s)) \circ \ldots \circ G'_{i_d}(G_{i_1 \cdots i_{d-1}}(s)).$$

So example x_s^i is equal to $i \circ G^{i_1 \cdots i_k}(s)$.

We now demonstrate an algorithm that on input x_s^i and $j > i$ computes z_s^j in time $O(nT_k)$ where T_k is the time to compute G on a k-bit input. The essential idea of the algorithm is that contained in example x_s^i are ancestors in the full binary tree of each piece of information contained in z_s^j. The reason we want to compute the entire z_s^j and not just the classification $c_s(x_s^j)$ is that the learning algorithm, given an example x that purports to be some x_s^j (because its first k bits are j), first will compute x_s^j to verify that x really is the "good example," and then predict positive exactly when both $x = x_s^j$ and the classification of x_s^j is 1.

ALGORITHM COMPUTE-FORWARD

Given: x_s^i and $j > i$.

1. Write $i = i_1 \cdots i_k$ and $j = j_1 \cdots j_k$. Let r be the least index such that $i_r \neq j_r$. Since $j > i$, we have $i_r = 0$ and $j_r = 1$.

2. Extract from x_s^i the portions:

$$u = G'_{i_1}(s) \circ G'_{i_2}(G_{i_1}(s)) \circ \ldots \circ G'_{i_{r-1}}(G_{i_1 \cdots i_{r-2}}(s)) \;=\; G^{i_1 \cdots i_{r-1}}(s).$$

$$v = G'_{i_r}(G_{i_1 \cdots i_{r-1}}(s)) \;=\; G_{j_1 \cdots j_r}(s).$$

(Notice that $G'_{j_r}(G_{j_1 \cdots j_{r-1}}(s)) = \lambda$.)

3. If $r = k$, output: $\langle j \circ u \circ \lambda, \mathrm{LSB}[v] \rangle$.
Otherwise, output: $\langle j \circ u \circ \lambda \circ G^{j_{r+1} \cdots j_k}(v), \mathrm{LSB}[G_{j_{r+1} \cdots j_k}(v)] \rangle$.

Algorithm Compute-Forward produces z_s^j as output for the following reason. By the definition of r, we know $G^{i_1 \cdots i_{r-1}}(s) = G^{j_1 \cdots j_{r-1}}(s)$, so $u \circ \lambda = G^{j_1 \cdots j_{r-1}}(s) \circ G'_{j_r}(G_{j_1 \cdots j_{r-1}}(s)) = G^{j_1 \cdots j_r}(s)$. Since $v = G_{j_1 \cdots j_r}(s)$, we have

$$u \circ \lambda \circ G^{j_{r+1} \cdots j_k}(v) \;=\; G^{j_1 \cdots j_r}(s) \circ G^{j_{r+1} \cdots j_k}(G_{j_1 \cdots j_r}(s)) \;=\; G^{j_1 \cdots j_k}(s),$$

and $\mathrm{LSB}[G_{j_{r+1} \cdots j_k}(v)] = \mathrm{LSB}[G_{j_1 \cdots j_k}(s)] = c_s(x_s^j)$. In addition, Algorithm Compute-Forward uses at worst $O(k^2)$ computations of G.

THEOREM 3.1. *Concept class $\mathcal{C}_G$ is learnable in the distribution-free model.*

Proof. Algorithm Learn-$\mathcal{C}_G$ works as follows. Collect $m \geq \frac{1}{\varepsilon} \ln \frac{1}{\delta}$ labeled examples. If all examples are negative, hypothesize the empty concept. Otherwise, we know each positive example is a "good example," so let i be the index (first k bits) of the positive example x_s^i of least index seen so far. Produce the following hypothesis: "On example x, let j be the first k bits of x. If $j < i$ predict 0. If $j > i$, use algorithm Compute-Forward to produce labeled example $\langle x_s^j, c_s(x_s^j) \rangle$. If $x = x_s^j$ then predict $c_s(x_s^j)$; otherwise predict 0."

One simple way to see that Learn-$\mathcal{C}_G$ learns in the distribution-free model is just to notice that it is an "Occam Algorithm" in the sense of Blumer, Ehrenfeucht, Haussler, and Warmuth [8] since for any size sample, the hypothesis produced has size $O(n)$ and is consistent with all the data seen. Alternatively, one can use a direct argument of the type used for learning a subinterval $[0, a]$ of the interval $[0, 1]$. Let l be the least index such that the set of examples $S_l = \{z_s^j : j \leq l \text{ and } c_s(x_s^j) = 1\}$ has probability at least ε. Algorithm Learn-$\mathcal{C}_G$ fails to learn with error less than ε exactly when it sees no example from S_l. So, the probability Learn-$\mathcal{C}_G$ fails is at most $(1 - \varepsilon)^m$ which is less than δ for $m \geq \frac{1}{\varepsilon} \ln \frac{1}{\delta}$. □

THEOREM 3.2. *Concept class $\mathcal{C}_G$ cannot be learned in the mistake-bound model if G is a CSB generator.*

In order to prove Theorem 3.2, we show that the sequence of examples x_s^i is difficult to compute in the reverse direction, as described in the following lemma and corollary.

LEMMA 3.3. *For any PPT Algorithm A and polynomial Q, for sufficiently large k, for all $i \in \{0, 1\}^k$,*

$$|\mathcal{P}_k(A, z_s^i) - \mathcal{P}_k(A, \bar{z}_s^i)| < \frac{1}{Q(k)}.$$

That is, for any Algorithm A and index i, for random seeds s, Algorithm A cannot distinguish between a correctly labeled x_s^i and an incorrectly labeled one. Using Algorithm Compute-Forward, on input x_s^i one can easily compute any other labeled example z_s^j for $j > i$. So, Lemma 3.3 could equivalently be written as the following.

COROLLARY 3.4. *For any PPT algorithm A and polynomial Q, for sufficiently large k, for all $i \in \{0, 1\}^k$,*

$$|\mathcal{P}_k(A^{\mathcal{O}}, z_s^i) - \mathcal{P}_k(A^{\mathcal{O}}, \bar{z}_s^i)| < \frac{1}{Q(k)},$$

where $\mathcal{O}$ is an oracle that on any input $j > i$ outputs labeled example z_s^j.

Lemma 3.3 implies, as shown below, that any learning algorithm will be fooled in the mistake-bound model by an adversary that presents the examples x_s^i in reverse order. In fact, Lemma 3.3 is stronger than we need because it implies that any algorithm will fail not just on *some* $c_s \in \mathcal{C}_G$ but on almost all of the concepts c_s.

Proof of Theorem 3.2 (assuming Lemma 3.3). Suppose there exists a learning algorithm L and polynomial P such that for all $c_s \in \mathcal{C}_G$ over $\{0, 1\}^n$, Algorithm L makes at most $P(n)$ mistakes. Consider an adversary that presents the good examples in reverse order: $x_s^{2^k}, x_s^{2^k-1}, \dots$. Algorithm L makes a mistake on at most $\frac{1}{4}$ of the first $4P(n)$ examples presented. If L is deterministic, this implies there exists some index t ($2^k - 4P(n) \leq t \leq 2^k$) such that over all $c_s \in \mathcal{C}_G$, Algorithm L makes an error an at most $\frac{1}{4}$ of the examples x_s^t. If L is randomized, we still have that for some t the probability over the choice of s and the coin tosses of L that L makes a mistake on x_s^t is at most $\frac{1}{4}$. So, we can use L in an Algorithm A that contradicts Corollary 3.4: On input $\langle x_s^t, b\rangle$ we use the oracle (or Algorithm Compute-Forward) to simulate the adversary presenting examples to L and then output 1 if L's prediction of $c_s(x_s^t)$ equals b. If b was the correct classification, there is at least a $\frac{3}{4}$ probability that L will output 1, but there is only a $\frac{1}{4}$ probability if b was incorrect. □

We now prove Lemma 3.3, showing that given only example x_s^i it is difficult to calculate the classification $c_s(x_s^i)$. The basic idea of the proof is a version of a standard cryptographic technique described by Yao [26]. Roughly, if an algorithm can distinguish strings z_s^i from strings $\overline{z}_s^i$, then we will slowly substitute bits in those strings with random bits and watch the performance of the algorithm degrade. At some point before the strings have become completely random the performance must degrade significantly, and we will focus on that location to break the generator.

Proof of Lemma 3.3. Suppose to the contrary there is a PPT Algorithm A and polynomial Q such that for infinitely many k, for some $t(k) \in \{0, 1\}^k$, we have $|\mathcal{P}_k(A, z_s^{t(k)}) - \mathcal{P}_k(A, \overline{z}_s^{t(k)})| \geq \frac{1}{Q(k)}$. Without loss of generality, let us assume that A has $\{0, 1\}$ output. We will use A to create an Algorithm B that breaks generator G. In particular, for infinitely many k, we will have $|\mathcal{P}_k(B, G(s)) - \mathcal{P}_{2k}(B, s)| \geq 1/2kQ(k)$.[2] Let $S_0' = \{0, 1\}^k$ and $S_1' = \{\lambda\}$ (to correspond to the notation for G_0' and G_1') and let $t = t(k) = t_1 \dots t_k$. Let $p_{1,k} = \mathcal{P}_k(A, z_s^t)$ and $\overline{p}_{1,k} = \mathcal{P}_k(A, \overline{z}_s^t)$, and for $1 < d \leq k$, define:

$$p_{d,k} = \mathbf{Pr}[A\langle t \circ r_1 \circ \dots \circ r_{d-1} \circ G^{t_d \cdots t_k}(s), b\rangle = 1 \mid r_1 \in_R S_{t_1}', \dots, r_{d-1} \in_R S_{t_{d-1}}',$$
$$s \in_R \{0, 1\}^k]$$

$$\overline{p}_{d,k} = \mathbf{Pr}[A\langle t \circ r_1 \circ \dots \circ r_{d-1} \circ G^{t_d \cdots t_k}(s), \overline{b}\rangle = 1 \mid r_1 \in_R S_{t_1}', \dots, r_{d-1} \in_R S_{t_{d-1}}',$$
$$s \in_R \{0, 1\}^k]$$

where $b = \mathrm{LSB}[G_{t_d \cdots t_k}(s)]$ and $\overline{b} = 1 - \mathrm{LSB}[G_{t_d \cdots t_k}(s)]$.

So, for $d > 1$, $p_{d,k}$ is the probability that A outputs 1 on input z_s^t where the first $d - 1$ "pieces" of x_s^t have been replaced by random strings of the appropriate length and the application of generator G begins at depth d in the full binary tree.

We have by assumption that $|p_{1,k} - \overline{p}_{1,k}| \geq 1/Q(k)$ for infinitely many k. We now consider two cases. The first is that $|p_{k,k} - \overline{p}_{k,k}| \geq 1/(kQ(k))$ for infinitely many k. The second is that

[2] Algorithm B will be nonuniform, in part because we have no guarantee that the index $t(k)$ is easy to compute given k (to contradict Lemma 3.3 just requires there exist *some* $t(k)$). One could modify (and make less "clean") the statement of Lemma 3.3 so that this problem no longer occurs and still separate the learning models, since our adversary chooses the examples in an easy to compute ordering.

for infinitely many k we have both $|p_{1,k} - \overline{p}_{1,k}| \geq 1/Q(k)$ and $|p_{k,k} - \overline{p}_{k,k}| < 1/(kQ(k))$. In the first case, the following Algorithm B will break generator G (the analysis follows the description of the algorithm).

ALGORITHM B

```
1. On input $y \in \{0,1\}^{2k}$, let $y_0$ be the left half of $y$ and $y_1$ be the right half of $y$. Let
   $y'_0 = y_1$ and $y'_1 = \lambda$.
2. For $i \in \{1, \ldots, k-1\}$ choose $r_i \in_R S'_{t_i}$.
3. Flip a coin:
      (i) If heads, output: $A\langle t \circ r_1 \circ \ldots \circ r_{k-1} \circ y'_{t_k}, \mathrm{LSB}[y_{t_k}]\rangle$.
      (ii) If tails, output: $1 - A\langle t \circ r_1 \circ \ldots \circ r_{k-1} \circ y'_{t_k}, 1 - \mathrm{LSB}[y_{t_k}]\rangle$.
```

If the input y to B equals $G(s)$ for random s, then the probability B outputs 1 is: $\frac{1}{2}p_{k,k} + \frac{1}{2}(1-\overline{p}_{k,k}) = \frac{1}{2}+\frac{1}{2}(p_{k,k}-\overline{p}_{k,k})$. If the input y to B is randomly chosen from $\{0,1\}^{2k}$, however, then we claim the probability B outputs 1 is just $\frac{1}{2}$. The reason is that since y_{t_k} and y'_{t_k} are disjoint pieces of y, B outputs with equal probability either $A\langle t \circ r, b\rangle$ or $1 - A\langle t \circ r, b\rangle$ for independent, random r and b. So, for infinitely many k we have $|\mathcal{P}_k(B, G(s)) - \mathcal{P}_{2k}(B, s)| \geq |\frac{1}{2}(p_{k,k} - \overline{p}_{k,k})| \geq 1/(2kQ(k))$; breaking G.

The second case, recall, is that for infinitely many k we have $|p_{1,k} - \overline{p}_{1,k}| \geq \frac{1}{Q(k)}$ and $|p_{k,k} - \overline{p}_{k,k}| < \frac{1}{kQ(k)}$. If k is such that these two inequalities hold, then either $|p_{d,k} - p_{d+1,k}| \geq 1/((2k-1)Q(k))$ or $|\overline{p}_{d,k} - \overline{p}_{d+1,k}| \geq 1/((2k-1)Q(k))$ for some $d \in \{1, 2, \ldots, k-1\}$; this just follows from the fact that $|p_{1,k} - \overline{p}_{1,k}| \leq |p_{1,k} - p_{2,k}| + \cdots + |p_{k-1,k} - p_{k,k}| + |p_{k,k} - \overline{p}_{k,k}| + |\overline{p}_{k,k} - \overline{p}_{k-1,k}| + \cdots + |\overline{p}_{2,k} - \overline{p}_{1,k}|$. So, this second case implies that either

(i) for infinitely many k there exists $d = d(k) \in \{1, \ldots, k-1\}$ such that $|p_{d,k} - p_{d+1,k}| > 1/(2kQ(k))$, or

(ii) for infinitely many k there exists $d = d(k) \in \{1, \ldots, k-1\}$ such that $|\overline{p}_{d,k} - \overline{p}_{d+1,k}| > 1/(2kQ(k))$.

Let us assume that the first situation occurs; the second situation is completely analogous. Algorithm B' to break generator G is then as follows.

ALGORITHM B′

```
1. On input $y \in \{0,1\}^{2k}$, let $y_0$ be the left half of $y$ and $y_1$ be the right half of $y$. Let
   $y'_0 = y_1$ and $y'_1 = \lambda$. Let $d = d(k)$ as above.
2. For $i \in \{1, \ldots, d-1\}$ choose $r_i \in_R S'_{t_i}$.
3. Output:
      $A\langle t \circ r_1 \circ r_2 \circ \ldots \circ r_{d-1} \circ y'_{t_d} \circ G^{t_{d+1} \cdots t_k}(y_{t_d}), \mathrm{LSB}[G_{t_{d+1} \cdots t_k}(y_{t_d})]\rangle$.
```

If the input y to B' equals $G(s)$ for random s, then $y'_{t_d} \circ G^{t_{d+1} \cdots t_k}(y_{t_d}) = G^{t_d \cdots t_k}(s)$ and $G_{t_{d+1} \cdots t_k}(y_{t_d}) = G_{t_d \cdots t_k}(s)$. So the probability B' outputs 1 is $p_{d,k}$. On the other hand, if $y \in_R \{0,1\}^{2k}$, then y_{t_d} and y'_{t_d} are completely independently chosen strings, so the probability B' outputs 1 is $p_{d+1,k}$. By the choice of d, Algorithm B' breaks the generator G on infinitely many k, a contradiction. □

So by Theorems 3.1 and 3.2, concept class C_G is learnable in the distribution-free model but not in the absolute mistake-bound model if G is a CSB generator, and such G exist if one-way functions exist.

4. Allowing membership queries. It turns out that a mistake-bound model learning algorithm can do no better on this concept class even if it is allowed to make membership queries. The reason is that in order for membership queries to help, the algorithm at some point must produce an entire good example x_s^i it has not yet seen to query. (Otherwise the membership query oracle could be replaced by a machine that always answers "negative" to any query of an example not yet seen.) As we show below, this would allow one to break G.

THEOREM 4.1. *Concept class $\mathcal{C}_G$ cannot be learned in the mistake-bound model with membership queries if G is a CSB generator.*

Proof. Suppose for contradiction there is a learning algorithm L and polynomial P such that for all $c_s \in \mathcal{C}_G$ over $\{0, 1\}^n$, Algorithm L makes at most $P(n)$ mistakes plus queries. The adversary will present the good examples in the order: $x_s^{2^k}, x_s^{2^k-1}, \ldots$, and let us consider the behavior of L on the first $5P(n)$ examples. Clearly, L makes a mistake on at most $\frac{1}{5}$ of the examples and makes at most $P(n)$ membership queries.

Let us say that a membership query is a "good query" if it is a query of a good example L has not yet seen. Also, let us define the algorithm *dequery*(L) to be the algorithm for the mistake-bound model (no queries) that runs L, and each time L attempts to make a membership query on an example it has not yet seen, returns to L the answer "no."

Suppose first that the probability that L makes no good queries during the first $5P(n)$ examples is at least $\frac{4}{5}$ (probability taken over the random choice of $c_s \in \mathcal{C}_G$ and the coin tosses of L). This would imply that for some index t, over random s, the algorithm *dequery*(L) has a probability at least $\frac{4}{5} - \frac{1}{5} = \frac{3}{5}$ of producing the correct classification of example x_s^t. This contradicts Corollary 3.4.

We may now assume that L makes a good query at some point in the first $5P(n)$ examples with probability at least $\frac{1}{5}$ (over random s). Thus, for some $t \geq 2^k - 5P(n)$, the probability algorithm L makes its *first* good query after seeing x_s^t but before seeing x_s^{t-1} is at least $1/(25P(n))$. Let $t = t_1 \ldots t_k$ and let $Q(n) = 25[P(n)]^2$.

Note that $G_{t_1 \cdots t_k}(s)$ is efficiently computable from any good example x_s^i for $i < t$. So, we can use L in an Algorithm A that on input x_s^t has a probability at least $\frac{1}{Q(n)}$ of producing $G_{t_1 \cdots t_k}(s)$ as output, as follows. Algorithm A uses x_s^t and Algorithm Compute-Forward to run L on inputs $x_s^{2^k}, x_s^{2^k-1}, \ldots, x_s^t$, returning "no" as an answer to each membership query made. It then guesses which membership query made after x_s^t is the good query (in case L makes several queries after x_s^t; the number of such queries is at most $P(n)$). A then uses that query to produce a hypothesis for $G_{t_1 \cdots t_k}(s)$.

In analogy to definitions in the proof of Lemma 3.3, let $q_{d,k}$ be the probability that A outputs $G_{t_d \cdots t_k}(s)$ on input $t \circ r_1 \circ \cdots \circ r_{d-1} \circ G^{t_d \cdots t_k}(s)$, where r_i is a randomly chosen string of length k if $t_i = 0$ and $r_i = \lambda$ if $t_i = 1$. We are given that $q_{1,k} \geq \frac{1}{Q(k)}$. However, we know that $q_{k,k}$ must be small for the following reason. Quantity $q_{k,k}$ is the probability that A outputs $G_{t_k}(s)$ on input $t \circ r \circ G'_{t_k}(s)$, where r is a random string of the appropriate length. If A does so with probability greater than $1/kQ(k)$, then Algorithm B to break G is as follows: on input y (let y_0 be the left half and y_1 be the right half), pick a random r and feed to A input $t \circ r \circ y_1$ if $t_k = 0$ or $t \circ r \circ \lambda$ if $t_k = 1$, and output 1 if A outputs y_{t_k}. If $y = G(s)$ for random s, then B outputs 1 with probability at least $1/kQ(k)$, but if y is a random string then B outputs 1 with probability at most $(1/2)^k$ since y_{t_k} is a random string of length k and independent of the input to A. Hence B breaks G.

Thus, there must exist some value d such that $|q_{d,k} - q_{d+1,k}| > 1/kQ(k)$. We can now break G with a variant of Algorithm B' of the proof of Lemma 3.3. On input $y \in \{0, 1\}^{2k}$ the algorithm outputs 1 if $A(t \circ r_1 \circ \cdots \circ r_{d-1} \circ y'_{t_d} \circ G^{t_{d+1} \cdots t_k}(y_{t_d})) = G_{t_{d+1} \cdots t_k}(y_{t_d})$. So if $y = G(s)$ for random s, it outputs 1 with probability $q_{d,k}$, and if $y \in_R \{0, 1\}^{2k}$, then it outputs 1 with probability $q_{d+1,k}$, and thus breaks G. □

5. Conclusion. We have shown how to construct a concept class over $\{0, 1\}^n$ that is learnable in the distribution-free model but not in the mistake-bound model if cryptographically secure bit generators (or equivalently one-way functions) exist. In fact, the assumption of one-way functions gives us something stronger. Not only is the concept class $\mathcal{C}_G$ hard to learn in the mistake-bound model in the sense that for any learning algorithm there is *some* $c \in \mathcal{C}_G$ hard to learn, but in fact we have that for any learning algorithm, nearly all $c \in \mathcal{C}_G$ are hard to learn. The fraction of concepts which an algorithm *can* learn is less than any polynomial fraction $\frac{1}{Q(n)}$ (for sufficiently large n). This fact leads one to ask the question: is it possible to separate the two models using a weaker assumption? Some assumption is apparently necessary since if computational considerations are ignored, any concept class learnable in one model is learnable in the other.

The concept class $\mathcal{C}_G$ remains difficult to learn in the mistake-bound model even if the learner is allowed membership queries. This lies in contrast to the class of DFAs which *are* learnable in the mistake-bound (equivalence query) model with membership queries [2] but not in the distribution-free model [19]. Thus, neither model is strictly easier than the other. Recently, membership queries have been shown not to help in learning (for a reason similar to that described here) for the important class of DNF formulas [4].

The concept class constructed here is very nonnatural. To date, every "natural" concept class known to be polynomial-time learnable in the distribution-free model is also known to be polynomial-time learnable in the mistake-bound model. However, for some of these classes the known mistake-bound algorithm is qualitatively more difficult and sometimes followed quite a bit later than the distribution-free algorithm. One such case is the class of decision lists (distribution-free algorithm by Rivest [23], mistake-bound algorithm by Littlestone (personal communication) and Helmbold, Sloan, and Warmuth [14]). With the addition of membership queries to both models, another case is that of read-twice DNF (distribution-free with queries algorithm by Hancock [10], and a more complicated mistake-bound with queries algorithm by Aizenstein and Pitt [1]). An interesting open problem is whether there is any natural class that would be a good candidate for being learnable in the distribution-free model but not the mistake-bound model.

Acknowledgments. I would like to thank Shafi Goldwasser, Silvio Micali, Ron Rivest, and Phil Rogaway for helpful discussions, and Rob Schapire for suggesting the simple "Occamness" argument for the proof of Theorem 3.1.

REFERENCES

[1] H. Aizenstein and L. Pitt, *Exact learning of read-twice DNF formulas*, Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, IEEE Computer Society Press, New York, pp. 170–179, 1991.

[2] D. Angluin, *Learning regular sets from queries and counterexamples*, Inform. and Comput., 75 (1987), pp. 87–106.

[3] ———, *Queries and concept learning*, Machine Learning, 2 (1988), pp. 319–342.

[4] D. Angluin and M. Kharitonov, *When won't membership queries help*, Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing, New Orleans, LA, ACM Press, New York, pp. 444–454, 1991.

[5] A. Blum, *Separating distribution-free and mistake-bound learning models over the Boolean domain*, Proceedings of the 31st Annual Symposium on Foundations of Computer Science, St. Louis, MO, IEEE Computer Society Press, New York, pp. 211–218, 1990.

[6] L. Blum, M. Blum, and M. Shub, *A simple unpredictable pseudo-random number generator*, SIAM J. Comput., 15 (1986), pp. 364–383.

[7] M. Blum and S. Micali, *How to generate cryptographically strong sequences of pseudo-random bits*, SIAM J. Comput., 13 (1984), pp. 850–863.

[8] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, *Occam's razor*, Inform. Process. Let., 24 (1987), pp. 377–380.

[9] O. Goldreich, S. Goldwasser, and S. Micali, *How to construct random functions*, Journal ACM, 33 (1986), pp. 792–807.

[10] T. Hancock, *Learning 2μ DNF formulas and $k\mu$ decision trees*, Proceedings of the Fourth Annual Workshop on Computational Learning Theory, Santa Cruz, CA, Morgan Kaufmann, San Mateo, CA, pp. 199–209, 1991.

[11] J. Hastad, *Pseudo-random generators under uniform assumptions*, Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing, Baltimore, MD, ACM Press, New York, 1990.

[12] D. Haussler, M. Kearns, N. Littlestone, and M. K. Warmuth, *Equivalence of models for polynomial learnability*, Proceedings of the 1988 Workshop on Computational Learning Theory, Cambridge, Morgan Kaufmann, San Mateo, CA, pp. 42–55, 1988.

[13] D. Haussler, N. Littlestone, and M. K. Warmuth, *Predicting* $\{0, 1\}$*-functions on randomly drawn points*, Proceedings of the Twenty-Ninth Annual Symposium on Foundations of Computer Science, White Plains, NY, IEEE Computer Society Press, New York, pp. 100–109, 1988.

[14] D. Helmbold, R. Sloan, and M. K. Warmuth, *Learning nested differences of intersection-closed concept classes*, Proceedings of the Second Annual Workshop on Computational Learning Theory, Santa Cruz, CA, Morgan Kaufmann, San Mateo, CA, pp. 41–56, 1989.

[15] R. Impagliazzo, L. A. Levin, and M. Luby, *Pseudo-random generation from one-way functions*, Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, Seattle, WA, ACM Press, New York, pp. 12–24, 1989.

[16] M. Kearns, *The Computational Complexity of Machine Learning*, Ph.D. thesis, Harvard University Center for Research in Computing Technology, Technical Report TR-13-89, Harvard Univ., Cambridge, MA, 1989.

[17] M. Kearns, M. Li, L. Pitt, and L. Valiant, *On the learnability of boolean formulae*, Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, New York, ACM Press, New York, pp. 285–295, 1987.

[18] ———, *Recent results on boolean concept learning*, Proceedings of the Fourth International Workshop on Machine Learning, Morgan Kaufmann, San Mateo, CA, pp. 337–352, 1987.

[19] M. Kearns and L. G. Valiant, *Cryptographic limitations on learning boolean formulae and finite automata*, Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, Seattle, WA, ACM Press, New York, pp. 433–444, 1989.

[20] L. A. Levin, *One-way functions and pseudorandom generators*, Proceedings of the Seventeenth ACM Symposium on Theory of Computing, Providence, RI, ACM Press, New York, pp. 363–365, 1985.

[21] N. Littlestone, *Learning when irrelevant attributes abound: A new linear-threshold algorithm*, Machine Learning, 2 (1988), pp. 285–318.

[22] W. Maass and G. Turán, *On the complexity of learning from counterexamples*, Proceedings of the Thirtieth Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, IEEE Computer Society Press, New York, pp. 262–267, 1989.

[23] R. L. Rivest, *Learning decision lists*, Machine Learning, 2 (1987), pp. 229–246.

[24] R. E. Schapire, *The strength of weak learnability*, Proceedings of the Thirtieth Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, IEEE Computer Society Press, New York, 1989.

[25] L. G. Valiant, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[26] A. C. Yao, *Theory and application of trapdoor functions*, Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science, Chicago, IL, IEEE Computer Society Press, New York, pp. 80–91, 1982.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 1001–1018, October 1994

COMPUTING WITH NOISY INFORMATION*

URIEL FEIGE†, PRABHAKAR RAGHAVAN‡, DAVID PELEG§ AND ELI UPFAL¶

Abstract. This paper studies the depth of *noisy* decision trees in which each node gives the wrong answer with some constant probability. In the *noisy Boolean decision tree* model, tight bounds are given on the number of queries to input variables required to compute threshold functions, the parity function and symmetric functions. In the *noisy comparison tree* model, tight bounds are given on the number of noisy comparisons for searching, sorting, selection and merging. The paper also studies parallel selection and sorting with noisy comparisons, giving tight bounds for several problems.

Key words. fault-tolerance, reliability, noisy computation, sorting and searching, error-correction

AMS subject classifications. 68M15, 68P10, 68R05

1. Introduction.

Fault-tolerance is an important consideration in large systems. Broadly, there are two approaches to coping with faults. The first is the "reconfiguration" approach [7], [13], in which faults are identified and isolated in real time. This is done concurrently with computation, and often has a significant overhead. A second, different approach is to devise robust algorithms that work despite unreliable information operations, without singling out the faulty components. This latter approach has been the focus of much recent work [11], [23], [16]–[19] [26], [22], [14], [15]. These papers differ in their general setting and in the mechanisms they use to model the faulty behavior of components. This paper concerns the probabilistic setting to this latter paradigm, as in [22], [14], and [15].

1.1. Model.

Our general model will be a (possibly randomized) computation tree, in which each node gives the correct answer with some probability, which is at least p, where p is a fixed constant in $(1/2, 1)$, bounded away from $1/2$ and 1. The node faults are independent. We study the depth of the computation tree in terms of a tolerance parameter $Q \in (0, 1/2)$: on any instance, the computation tree must lead to a leaf giving the correct answer on that instance with probability at least $1 - Q$. The success probability of the algorithm is computed over the combined probability space of the outcome of individual operations and the results of coin flips (in case our algorithm is randomized).

There are several possible types of computation trees that could be studied in this noisy tree model; this paper focuses on two. The first is the *noisy Boolean decision tree*, in which the tree computes a Boolean function of N Boolean variables $x_1, \dots, x_N$. Each node in the tree corresponds to querying one of the input variables; with some probability, we are given the wrong value of that variable. Each leaf is labeled 0 or 1, and corresponds to an evaluation of the function.

The second type studied is *noisy comparison trees* for problems such as sorting, selection, and searching. Here the input is a set $\{x_1, x_2, \dots, x_N\}$ of N numbers. (For searching, the input contains also x_{-1}, the searched element.) Each node in the tree specifies two indices i and j of the elements to be compared. (In our searching algorithms, for instance, one of these indices is always the searched element.) The node responds with either "$x_i \geq x_j$" or

*Received by the editors March 4, 1991; accepted for publication (in revised form) June 8, 1993.

†The Weizmann Institute of Science, Rehovot, Israel. Part of the work was done while this author was visiting IBM T.J. Watson and Almaden Research Centers, San Jose, California 95120.

‡IBM T.J. Watson Research Center, Yorktown Heights, New York 10598. A portion of this work was done while the author was visiting the Weizmann Institute of Science, Rehovot, Israel.

§The Weizmann Institute of Science, Rehovot, Israel. The work of this author was supported in part by an Allon Fellowship, a Bantrell Fellowship and a Walter and Elise Haas Career Development Award.

¶IBM Almaden Research Center, San Jose, California 95120. This author's work at the Weizmann Institute was supported in part by a Bat-Sheva de Rothschild Award and by a Revson Career Development Award.

"$x_i \leq x_j$," and gives the wrong answer with some probability. Each leaf is labeled with a permutation representing the sorted order for the input (for sorting and merging) or an index in $[1, N]$ (for selection and searching).

A simple example is in order here. In the absence of errors, the maximum of N numbers can be found by a comparison tree of depth $N - 1$. In the face of a constant probability of error, it is possible to repeat each comparison of the fault-free decision tree $O(\log(N/Q))$ times and obtain, by majority voting, a guess for the true result of the comparison that is wrong with probability at most Q/N. (Note that this does not require a special majority operation, but only "blowing up" each node of the tree into a subtree of the appropriate depth.) Doing this for every comparison immediately gives us a noisy comparison tree of depth $O(N \log(N/Q))$ for finding the maximum (we can afford to sum the failure probability of Q/N over the $N - 1$ events). In a similar fashion, any decision tree that has depth d in the absence of noise can be used to devise a noisy one of depth $O(d \log(d/Q))$.

The crux of our work is to show that while this logarithmic blowup is unavoidable for certain problems, it is (perhaps surprisingly) unnecessary for certain others. In fact, we are able to show such a separation between problems that have the same decision tree complexity in the absence of errors, such as the threshold function with various parameters (Theorems 2.2 and 2.7). A major obstacle to proving the lower bounds is that *errors cancel*—multiple errors could compound on an input to lead to a leaf giving the correct answer to that input, for the "wrong reason."

Another distinction we make is between a *static adversary*, where the probability of correctness of every node of the tree is fixed at p, and a *dynamic adversary* who can set the probability of correctness of each tree node to any value in $[p, 1)$.

It turns out that there is a difference between these two cases. In the dynamic case, the noisy decision tree complexity is bounded below by the deterministic (noise free) decision tree complexity, since the adversary may always opt for a correct execution (with all individual operations giving the correct value). In contrast, in the static case, the noisy decision tree complexity is bounded above by $\log(n/Q)$ times the *randomized* noise free decision tree complexity. This follows from the fact that the availability of basic operations with fixed success probability provides us with a fixed-bias coin, which in turn can be used to generate a fair coin. Since there are problems for which the randomized decision tree complexity is significantly smaller than the deterministic decision tree complexity (cf. [24]), it follows that the presence of fixed probability faults may actually *help* the algorithm. This points out another source of difficulty in proving lower bounds in the noisy decision tree model.

1.2. Related previous work. Noisy comparison trees for binary search and related problems were studied by Renyi [22] and by Pelc [14], [15]. Pippenger [16] and others have studied networks of noisy gates, in which every gate could give the wrong answer with some probability. Kenyon-Mathieu and Yao [11] study a Boolean decision tree in which an adversary is allowed to corrupt at most k nodes (read operations) along any root-leaf path. Rivest et al. [23] consider the problem of binary search on N elements using a comparison tree when an adversary can choose k comparisons to be incorrect ("lies") on any root-leaf path. This model was further studied by Ravikumar et al. [18], [19]. Yao and Yao [26] study sorting networks with at most k faulty comparators.

Our work differs from [11], [23], [18], [19] in that we allow every node of the decision tree to be independently faulty with some probability. Thus in our model the number of faults is not prescribed in advance—knowledge of this number could well be exploited by a "fault-tolerant" algorithm. The probabilistic model allows us to tolerate a relatively large number of faults compared to [11], [23], [18], [19].

1.3. Results. Let $D^{\text{Prob}}_{N,Q}(\Pi)$ (respectively, $D^{\text{Det}}_{N,Q}(\Pi)$) denote the minimum depth of any noisy probabilistic (respectively, deterministic) decision tree for instances of size N of problem Π, with tolerance Q. For notational simplicity, we shall write $D_{N,Q}(\Pi) = \Theta(f)$ to denote both $D^{\text{Prob}}_{N,Q}(\Pi) = \Omega(f)$ and $D^{\text{Det}}_{N,Q}(\Pi) = O(f)$.

All our lower bounds are for probabilistic trees, and all the upper bounds (with the exception of parallel sorting) are for deterministic trees. Furthermore, all our lower bounds apply against the weaker static adversary (and hence also against a dynamic adversary), and all the upper bounds apply against a dynamic adversary (and hence also against a static adversary). Since all of our bounds are tight (up to constant factors), we conclude that randomization does not significantly help for the problems studied (with the possible exception of parallel sorting).

For any problem Π, the depth of its optimal decision tree is at most polynomial in the length of the input. However, the size of the decision tree is often exponential. An important feature of our upper bounds is that the corresponding decisions trees have descriptions which are polynomial in the length of the input. At any time step, the next query (or comparison) to be made is a simple function (i.e., computable in polynomial time) of the outcomes of the previous queries.

Let TH^N_K denote the K-of-N threshold function: given N Boolean inputs, the output is 1 if and only if K or more of the inputs are 1. The PARITY function on N Boolean inputs outputs 1 if and only if the number of 1's in the input is even. For noisy Boolean decision trees we have the following results (in §2).

(1) $D_{N,Q}(\text{TH}^N_K) = \Theta(N\log(m/Q))$, where $m = \min\{K, N-K\}$. In particular, $D^{\text{Det}}_{N,Q}(\text{OR})$ and $D^{\text{Det}}_{N,Q}(\text{AND})$ are both $O(N\log(1/Q))$.

(2) $D_{N,Q}(\text{PARITY}) = \Theta(N\log(N/Q))$.

Notice the wide range of noisy tree depths in these results, whereas in the absence of noise, decision trees for all these problems have depth N. Problems such as parity have a blowup in tree depth that grows with N, rather than p or Q alone (unlike the OR function). In §2 we extend these results to all symmetric functions.

Let K-SEL be the problem of selecting the Kth largest of N elements. In the noisy comparison tree model we have the following tight results (in §3).

(1) $D_{N,Q}(\text{BINARY SEARCH}) = \Theta(\log(N/Q))$.

(2) $D_{N,Q}(\text{SORTING}) = \Theta(N\log(N/Q))$.

(3) $D_{N,Q}(\text{MERGING}) = \Theta(N\log(N/Q))$.

(4) $D_{N,Q}(K\text{-SEL}) = \Theta(N\log(m/Q))$, where $m = \min\{K, N-K\}$.

In particular, the maximum or the minimum element can be found by a noisy tree of depth $O(N\log(1/Q))$.

A well-known sports commentator has observed [9] that the problem of finding the maximum by a noisy comparison tree has a sporting interpretation: we wish to find the best of N teams by a tournament. In each game, the better team wins with some probability, which is at least p; how many games must be played in order that the best team wins with probability at least $1-Q$? One algorithm we give for finding the maximum by a noisy comparison tree bears a remarkable resemblance to the NBA championship: teams pair up and play a game at the first round, the winners pair up and play three games at the next, five in the third round and so on. It can be shown that the best team fails to win such a tournament with probability at most $c'(1-p)$ for some c', and that the total number of games is $O(N)$. This failure probability can be reduced to Q by multiplying the number of games in each round by $c\log(1/Q)$.

This brings up the following natural question: how many days must such a tournament last, assuming a team plays at most one game a day? Similarly, what is the depth of a noisy "EREW" parallel comparison tree with up to $N/2$ parallel comparisons at each node? The "NBA" algorithm described above requires $\Theta(\log N\log(N/Q))$ rounds.

In §4 we show that $O(\log(N/Q))$ rounds suffice for this problem, while keeping the total number of games down to $O(N\log(1/Q))$. More precisely, we show that there is an N-processor EREW-PRAM algorithm that computes the maximum of N elements with noisy comparisons, using $O(\log(N/Q))$ rounds and a total of $O(N\log(1/Q))$ comparisons, with failure probability at most Q. The algorithm applies even when each element is allowed to participate in at most one comparison per round (i.e., no element duplication is allowed).

In §5 we give a randomized parallel algorithm for sorting. The algorithm is based on a randomized, noisy, parallel comparison tree (with N comparisons per node) of depth $O(\log N)$. For sorting N numbers, the failure probability of the algorithm can be made as small as $1-N^{-c}$ for any constant $c > 0$.

2. Boolean decision trees. The main result of this section is a lower bound on the depth of any noisy Boolean decision tree computing the K-of-N threshold function TH_K^N. As a first step, we prove a lower bound for the case $K = 1$, which is the OR function.

THEOREM 2.1. $D_{N,Q}^{\mathrm{Prob}}(\mathrm{OR}) = \Omega((N\log\frac{1-Q}{Q})/(\log\frac{p}{1-p}))$.

Proof. Let $\bar{X} = (X_1, \ldots, X_N)$ be the input vector, let $\bar{0} = (0, \ldots, 0)$ and let $\bar{1}_j$ denote an input vector $X_j = 1$ and the remaining inputs zero. The proof is based on showing that distinguishing between $\bar{0}$ and the adjacent vectors $\bar{1}_j$ requires the stated depth. For a leaf ℓ of a Boolean decision tree of depth d and an input vector $\bar{X}$, let $Pr\{\ell|\bar{X}\}$ denote the probability of reaching ℓ (in a probabilistic decision tree it combines the probabilities of the random choices of the algorithm with the probabilities of the random answers to the queries) on an input $\bar{X}$.

Assume that X_j appears $r(j, \ell)$ times on the path from the root to ℓ. Then

$$Pr\{\ell|\bar{1}_j\} \geq \left(\frac{1-p}{p}\right)^{r(j,\ell)} Pr\{\ell|\bar{0}\}.$$

For any ℓ, $\sum_{j=1}^N r(j,\ell) = d$. Therefore $\sum_{j=1}^N((1-p)/p)^{r(j,\ell)}$ achieves its minimum (over all choices of $r(j,\ell)$) at $N((1-p)/p)^{d/N}$.

For a set L of leaves, define

$$Pr\{L|\bar{X}\} = \sum_{\ell\in L} Pr\{\ell|\bar{X}\}.$$

Thus, letting S denote the set of leaves labeled 0, we get

$$\begin{aligned}\sum_{j=1}^N Pr\{S|\bar{1}_j\} &= \sum_{j=1}^N\sum_{\ell\in S} Pr\{\ell|\bar{1}_j\}\\ &\geq \sum_{\ell\in S}\sum_{j=1}^N\left(\frac{1-p}{p}\right)^{r(j,\ell)} Pr\{\ell|\bar{0}\}\\ &\geq Pr\{S|\bar{0}\}N\left(\frac{1-p}{p}\right)^{d/N}.\end{aligned}$$

Clearly $Pr\{S|\bar{0}\} \geq (1-Q)$, and for every j, $Pr\{S|\bar{1}_j\} \leq Q$, and hence

$$QN \geq (1-Q)N\left(\frac{1-p}{p}\right)^{d/N}.$$

The bound on d follows.

Note that the proof works with a static adversary. A somewhat simpler proof can be given if the adversary is dynamic (Theorem 4.1). □

Let us now turn to the general threshold function TH_K^N. For a vector $\bar{X} = (X_1, \ldots, X_N)$ of N bits, let $\omega(\bar{X})$ denote the *weight* of $\bar{X}$, i.e., $\omega(\bar{X}) = \sum_{i=1}^N X_i$. Thus

$$\mathrm{TH}_K^N = \begin{cases} 1, & \omega(\bar{X}) \geq K, \\ 0, & \text{otherwise.} \end{cases}$$

THEOREM 2.2. *For every* $K < N/2$,

$$D_{N,Q}^{\mathrm{Prob}}(\mathrm{TH}_K^N) = \Omega\left(\theta N \log K + \frac{N \log \frac{1-Q}{Q}}{\log \frac{p}{1-p}}\right),$$

for $\theta = (1 - \frac{Q}{1-Q})/(\log 1/(1-p))$.

We first give a high level overview of the main ideas of the proof.

The main difficulty in proving lower bounds in our model stems from the fact that algorithms may be adaptive. For our lower bound, it suffices to use a static adversary: each query has a fixed probability p of giving the right answer.

Let T be a noisy decision tree (algorithm) of depth γN that computes TH_K^N, where γ may be a function of N. Now we strengthen the algorithm, but make its adaptive behavior easier to analyze, by transforming it to a two-phase algorithm T_1 in a "more powerful" model. The transformation is based on the observation that in any execution of T, at most $N/3$ input variables are each queried more than $\alpha = 3\gamma$ times. (The choice of $1/3$ is somewhat arbitrary, and is replaced by the parameter μ later.)

(A) Nonadaptive phase: Query each variable exactly α times. Each query returns the correct value with probability p.

(B) Adaptive phase: Request the values of $N/3$ of the input variables. These requests are answered correctly. At each point, T_1's choice of which variable to read next may depend upon all the answers up to that point.

Since we are considering static adversaries and randomized algorithms, T_1 can simulate the execution of T. The algorithm T_1 first runs the nonadaptive phase (A), querying each input variable α times. In phase (B) it starts simulating the execution of T. As long as T queries a variable fewer than α times, T_1 supplies the answers from the answers it got in phase (A) on queries to that variable. Once T queries a variable more than α times (note that this may happen for at most $N/3$ of the variables), T_1 requests the (correct) value of this variable in the adaptive phase (B). It then uses this value to answer T's subsequent queries after corrupting it randomly with probability p. Clearly, on any input, the probability distribution on T_1's outputs is identical to that on T's outputs. Noting that the depth of T_1 is at most a constant times the depth of T plus N, any lower bound on the depth of T_1 implies a corresponding lower bound for T.

We now outline the approach to proving that TH_K^N cannot be computed reliably by a T_1 type algorithm if γ is $o(\log K)$. For the lower bound, we only supply inputs to the algorithm with weights either $K - 1$ (for which the algorithm should output 0) or K (for which the algorithm should output 1). We show that if T_1 has insufficient depth, it is unlikely to distinguish between inputs from these different weights (and thus output values).

Since phase (A) of T_1 is nonadaptive, it is relatively easy to analyze its outcome. We view this phase as a game of randomly placing N balls, $K - 1$ or K of which are black and the rest white, into $\alpha + 1$ bins, numbered 0 to α. A white (respectively, black) ball i corresponds to an input variable X_i that is set to 0 (respectively, 1). Ball i is placed in bin j if exactly j of the α queries to X_i were answered 1. (By symmetry, it suffices to count the number of 1 answers and ignore the ordering between them and the 0 answers.) The vector $(s_0, \ldots, s_\alpha)$, where s_j

is the number of balls in bin j, is called the *execution profile* of the nonadaptive phase (A). If $K > 1/(1-p)^{\alpha}$, each bin can be expected to have at least one ball of each color.

At the end of phase (A), T_1 gets to see its profile, but not the actual colors of the balls in the bins. Its must determine the number of black balls using noiseless queries to $N/3$ of the balls in phase (B), together with the execution profile from phase (A).

We employ one final device to simplify the analysis of phase (B). Before phase (B) begins, we "help" the algorithm T_1 by revealing the values "for free" of $K-1$ of the black balls (creating a new profile). In particular, if the input contained K black balls, then the single black ball to be left hidden, is chosen randomly with the probability distribution of the *white* balls. Now, if there were just $K-1$ black balls, then phase (B) will reveal only white balls, and if there were K black balls, then the probability that phase (B) reveals the remaining ball is only constant (bounded from above by $N/3(N-K+1) < 2/3$). Thus with constant probability, phase (B) gives T_1 no additional information about the number of black balls. In this case, T_1 must base its decision upon only two profiles seen, both of which were seen before phase (B) of the algorithm has begun. Thus we reduce our problem to the analysis of simple random allocation games. Now standard probability theory can be used to show that the distribution of profiles that result from inputs having $K-1$ black balls is statistically similar to the one that results from inputs having K black balls, making it impossible for T_1 to achieve a success probability better than some fixed constant bounded away from 1.

We turn to a detailed proof of the theorem.

Proof of Theorem 2.2. If $K \leq \max\{(1-Q)/Q, C\}$, for some constant C, then the adversary can announce the values of input variables $X_1, X_2, \ldots X_{K-1}$ in advance to be 1. Computing TH_K^N is then reduced to the problem of computing the OR function of the remaining $N-K+1$ bits. By Theorem 2.1, this requires a tree of depth

$$\Omega\left(\frac{N\log((1-Q)/Q)}{\log(p/(1-p))}\right).$$

Thus for the rest of the proof assume that $K > \max\{(1-Q)/Q, C\}$, for a sufficiently large constant C. Fix constants μ, θ such that

$$0 < \mu < \frac{1}{2}\left(1-\frac{Q}{1-Q}\right)$$

and

$$0 < \theta \leq -\frac{\mu}{3\log(1-p)}.$$

Given a noisy decision tree for TH_K^N of depth $\gamma \leq \theta N \log K$, we show that its failure probability exceeds Q, and this will yield the lower bound of $\theta N \log K$ on the depth.

Let

$$\alpha = -\frac{\log K}{3\log(1-p)}.$$

Since $\gamma \leq \theta N \log K$, the number of input variables that are queried more than α times in any particular computation path is at most

$$\frac{\gamma}{\alpha}.$$

By the previous discussion we may, without loss of generality, grant the decision tree some additional information as outlined, and prove the lower bound for trees of the following two-phase form, which are more powerful than any decision tree of depth $\gamma \leq \theta N \log K$.

Phase (A). Query every variable exactly α times. Each query returns the correct value with probability p.

Phase (B). Request the values of μN of the input variables $X_1, \ldots X_N$; these queries are answered correctly.

The proof is by means of the probabilistic method. We present the tree with randomly chosen inputs having $K-1$ or K ones. We show that certain leaves of the tree will be reached with almost the same probability regardless of whether the input has $K-1$ or K ones. We first analyze the outcome of Phase (A). Let S_i denote the set of variables, of whose α queries, i were answered 1 in Phase (A). The outcome of Phase (A) is fully characterized by the vector $\bar{S} = (S_0, \ldots, S_\alpha)$. Let $s_i = |S_i|$, and $\bar{s} = (s_0, \ldots, s_\alpha)$ (the execution profile of Phase (A)). Let z_i (respectively, y_i) denote the number of input variables in S_i whose actual values are 0 (respectively, 1). Let $\bar{z} = (z_0, \ldots, z_\alpha)$ and $\bar{y} = (y_0, \ldots, y_\alpha)$.

For a variable X, let

$$P_i^0 = Pr\{X \in S_i | X = 0\} = \binom{\alpha}{i} p^{\alpha-i}(1-p)^i.$$

Note that the expected value of z_i over the inputs of interest is either $(N-K)P_i^0$ or $(N-K+1)P_i^0$.

LEMMA 2.3. *$Pr\{\exists i, y_i = 0\} \leq 1/N$.*

Proof.

$$Pr\{\exists i, y_i = 0\} \leq \sum_{i=1}^{\alpha}(1-P_i^0)^{N-K}.$$

Since the minimum value of P_i^0 is obtained when $i = \alpha$,

$$Pr\{\exists i, y_i = 0\} \leq (\alpha+1)(1-P_\alpha^0)^{N-K} \leq (\alpha+1)(1-(1-p)^{\frac{-\log K}{3\log(1-p)}})^{N-K} \leq 1/N,$$

for sufficiently large N. □

We now argue that the probability of getting a vector $\bar{S}$ given $\omega(\bar{X}) = K$ and given $\omega(\bar{X}) = K-1$ are very close to each other. In order to prove that, we shall restrict our attention to the case when none of the z_i variables diverges from its expected value by much more than the standard deviation. Formally, let $\mathcal{E}$ be the event that for all $1 \leq i \leq \alpha$,

$$(N-K)P_i^0(1-\Delta_i) \leq z_i \leq (N-K)P_i^0(1+\Delta_i),$$

where

$$\Delta_i = 6\sqrt{\frac{\log N}{(N-K)P_i^0}}.$$

By the Chernoff bound [4], it follows that the case we focus on is the dominant one.

LEMMA 2.4. *$Pr\{\bar{\mathcal{E}}\} \leq 1/N$, where $\bar{\mathcal{E}}$ is the complement of $\mathcal{E}$.* □

We now derive the following lemma.

LEMMA 2.5.

$$1 - \frac{1}{N^{1/5}} \leq \frac{Pr\{\bar{S}|(\omega(\bar{X}) = K) \wedge \mathcal{E}\}}{Pr\{\bar{S}|(\omega(\bar{X}) = K-1) \wedge \mathcal{E}\}} \leq 1 + \frac{1}{N^{1/5}}.$$

Proof. Let $\bar{s}(\ell) = (s_0, \ldots, s_\ell - 1, \ldots, s_\alpha)$ denote a distribution of $N-1$ variables ($K-1$ ones and $N-K$ zeros) among the sets $S_0, \ldots, S_\alpha$. Similarly, denote by $\bar{z}(\ell) = (z_0, \ldots, z_\ell - 1, \ldots, z_\alpha)$ a distribution of the $N-K$ 0-variables among $S_0, \ldots, S_\alpha$.

For any ℓ_1, ℓ_2,

$$\frac{Pr\{\bar{z}(\ell_1)|\mathcal{E}\}}{Pr\{\bar{z}(\ell_2)|\mathcal{E}\}} = \frac{\binom{N-K}{z_1,\ldots,z_{\ell_1}-1,\ldots,z_\alpha}\prod_{i=1}^{\alpha}(P_i^0)^{z_i}\cdot\frac{1}{P_{\ell_1}^0}}{\binom{N-K}{z_1,\ldots,z_{\ell_2}-1,\ldots,z_\alpha}\prod_{i=1}^{\alpha}(P_i^0)^{z_i}\cdot\frac{1}{P_{\ell_2}^0}} = \frac{z_{\ell_1}P_{\ell_2}^0}{z_{\ell_2}P_{\ell_1}^0}.$$

By condition $\mathcal{E}$,

$$\frac{1-\Delta_{\ell_1}}{1+\Delta_{\ell_2}} \le \frac{z_{\ell_1}P_{\ell_2}^0}{z_{\ell_2}P_{\ell_1}^0} \le \frac{1+\Delta_{\ell_1}}{1-\Delta_{\ell_2}}.$$

Since for any i, $K^{-1/3} \le P_i^0 \le 1$,

$$\Delta_m = 6\sqrt{\frac{\log N}{N}} \le \Delta_i \le 6\sqrt{\frac{2\log N}{N^{2/3}}} = \Delta_M.$$

Thus,

$$1 - \frac{1}{N^{1/5}} \le 1 - \frac{2\Delta_m}{1+\Delta_m} \le \frac{z_{\ell_1}P_{\ell_2}^0}{z_{\ell_2}P_{\ell_1}^0} \le 1 + \frac{2\Delta_M}{1-\Delta_M} \le 1 + \frac{1}{N^{1/5}}.$$

By summing over all possible pairs $(\bar{z}(\ell), \bar{y})$ such that $\bar{z}(\ell) + \bar{y} = \bar{s}(\ell)$ we get

$$\frac{Pr\{\bar{s}(\ell_1)|\mathcal{E}\}}{Pr\{\bar{s}(\ell_2))|\mathcal{E}\}} = \frac{\sum_{\bar{z}(\ell_1)+\bar{y}=\bar{s}(\ell_1)} Pr\{\bar{z}(\ell_1)|\mathcal{E}\}Pr\{\bar{y}|\mathcal{E}\}}{\sum_{\bar{z}(\ell_2)+\bar{y}=\bar{s}(\ell_2)} Pr\{\bar{z}(\ell_2)|\mathcal{E}\}Pr\{\bar{y}|\mathcal{E}\}}.$$

Thus,

$$1 - \frac{1}{N^{1/5}} \le \frac{Pr\{\bar{s}(\ell_1)|\mathcal{E}\}}{Pr\{\bar{s}(\ell_2))|\mathcal{E}\}} \le 1 + \frac{1}{N^{1/5}}.$$

We now add the last variable X, with value either one or zero, and get

$$\frac{Pr\{\bar{s}|(\omega(\bar{X}) = K) \wedge \mathcal{E}\}}{Pr\{\bar{s}|(\omega(\bar{X}) = K-1) \wedge \mathcal{E}\}} = \frac{\sum_{\ell=1}^{\alpha} Pr\{\bar{s}(\ell)|\mathcal{E}\}Pr\{X \in S_\ell | X = 1\}}{\sum_{\ell=1}^{\alpha} Pr\{\bar{s}(\ell)|\mathcal{E}\}Pr\{X \in S_\ell | X = 0\}}.$$

Using the fact that $Pr\{X \in S_\ell | X = 0\} = Pr\{X \in S_{\alpha-\ell} | X = 1\}$ we have

$$1 - \frac{1}{N^{1/5}} \le \frac{Pr\{\bar{s}|(\omega(\bar{X}) = K) \wedge \mathcal{E}\}}{Pr\{\bar{s}|(\omega(\bar{X}) = K-1) \wedge \mathcal{E}\}} \le 1 + \frac{1}{N^{1/5}}.$$

Since all assignments of K or $K-1$ ones to the variables have equal probability, by symmetry, all partitions of the variables into sets of sizes $s_1, \ldots, s_\alpha$ have equal probability and the claim is proven. □

To simplify the analysis of Phase (B) we assume without loss of generality that at the end of Phase (A) the adversary reveals the locations of $K-1$ input variables with value 1. If

$\omega(\bar{X})$ is K, and for all i, $y_i > 0$, the remaining (unexposed) variable is chosen to be from set S_i with probability $z_i/(\sum_{j=1}^{\alpha} z_j) = z_i/(N-K)$.

Denote by $\tilde{S_1}, \ldots, \tilde{S_\alpha}$ the input to Phase (B), where $\tilde{S_i}$ contains the variables in S_i that were not revealed by the adversary. Note that $z_i/|\tilde{S_i}| \leq 1$ for all i (more specifically, $|\tilde{S_i}| = z_i$ for all sets S_i except at most one, which might have $z_i + 1$ variables).

The tree cannot distinguish between variables in $\tilde{S_i}$. Suppose that the tree queries r_i variables in $\tilde{S_i}$ ($\sum_i r_i = \mu N$). If $\tilde{S_i}$ contains the unexposed 1-variable, the probability that the tree hits the unexposed 1-variable is $r_i/|\tilde{S_i}|$. The probability that the 1-variable is in $\tilde{S_i}$ is proportional to $|\tilde{S_i}|$. Thus the probability P_{hit} of hitting the 1-variable in Phase (B) when for all i, $y_i > 0$ is bounded above by

$$(*) \qquad P_{hit} \cdot \frac{r_i}{|\tilde{S_i}|}.$$

Let $\mathcal{S}$ denote the event that the output of Phase (A) is a vector $\bar{S}$ with the property that if Phase (B) does not find a 1-variable, the algorithm outputs 0.

LEMMA 2.6.

$$Pr\{\mathcal{S}|\omega(\bar{X}) = K\} \geq \left(1 - \frac{1}{N^{1/5}}\right)(1-Q) - \frac{1}{N}.$$

Proof. Clearly $Pr\{\mathcal{S}|\omega(\bar{X}) = K-1\} \geq 1-Q$, or else the tree does not perform as claimed. Now

$$Pr\{\mathcal{S}|\omega(\bar{X}) = K-1\} \leq Pr\{\bar{\mathcal{E}}\} + Pr\{\mathcal{S}|(\omega(\bar{X}) = K-1) \wedge \mathcal{E}\}.$$

By Lemma 2.5,

$$\begin{aligned} Pr\{\mathcal{S}|\omega(\bar{X}) = K-1\} &\leq Pr\{\bar{\mathcal{E}}\} + Pr\{\mathcal{S}|(\omega(\bar{X}) = K) \wedge \mathcal{E}\}\left(1 + \frac{1}{N^{1/5}}\right) \\ &\leq Pr\{\bar{\mathcal{E}}\} + \frac{Pr\{\mathcal{S}|\omega(\bar{X}) = K\}}{Pr\{\mathcal{E}\}}\left(1 + \frac{1}{N^{1/5}}\right). \end{aligned}$$

Thus, for an input $\bar{X}$ with $\omega(\bar{X}) = K$, by Lemma 2.4,

$$\begin{aligned} Pr\{\mathcal{S}|\omega(\bar{X}) = K\} &\geq \frac{(1-\frac{1}{N})}{(1+\frac{1}{N^{1/5}})}\left(Pr\{\mathcal{S}|\omega(\bar{X}) = K-1\} - \frac{1}{N}\right) \\ &\geq \left(1 - \frac{1}{N^{1/5}}\right)\left(1 - Q - \frac{1}{N}\right), \end{aligned}$$

implying the lemma. □

We are now ready to complete the proof of Theorem 2.2. By $(*)$, when the output of Phase (A) satisfies event $\mathcal{S}$, $\omega(\bar{X}) = K$, and for all i, $y_i > 0$, the probability that Phase (B) finds the unexposed 1 is at most 2μ. Otherwise the algorithm outputs 0. Since $Q < 1/2$, when $\omega(\bar{X}) = K$, by Lemma 2.3,

$$Pr\{\exists i : y_i = 0|\mathcal{S}\} \leq \frac{Pr\{\exists i : y_i = 0\}}{Pr\{\mathcal{S}\}} \leq \frac{2}{N}.$$

Thus, the probability that the algorithm fails and outputs 0 when $\omega(\bar{X}) = K$ is at least

$$Pr\{\mathcal{S}|\omega(\bar{X}) = K\}(1 - Pr\{\exists i : y_i = 0|\mathcal{S}\})(1 - 2\mu) > Q$$

for sufficiently large N. □

A matching upper bound for TH_K^N follows from a variant of the algorithm for K-SEL in §3. We defer the details of the algorithm to §3. Thus we have a deterministic upper bound matching the lower bound for randomized algorithms.

THEOREM 2.7. *$D_{N,Q}^{\mathrm{Det}}(\mathrm{TH}_K^N) = O(N \log(m/Q))$, where $m = \min\{K, N-K\}$. In particular, $D_{N,Q}^{\mathrm{Det}}(\mathrm{OR})$ and $D_{N,Q}^{\mathrm{Det}}(\mathrm{AND})$ are both $O(N \log(1/Q))$.*

In fact, the algorithm for K-SEL implies a more general result. A Boolean function f on N Boolean variables is *symmetric* if $f(X_1, \ldots, X_N) = f(X_{\pi(1)}, \ldots, X_{\pi(N)})$ for every permutation π on $\{1, \ldots, N\}$. For the function f, let k_1 be the largest $i < N/2$ such that there exist $\bar{x}$ with $\omega(\bar{X}) = i - 1$, and $\bar{X}'$ with $\omega(\bar{X}') = i$, and $f(\bar{X}) \neq f(\bar{X}')$. Similarly, let k_2 be the smallest $i \geq N/2$ such that there exist $\bar{x}$ with $\omega(\bar{X}) = i$, and $\bar{X}'$ with $\omega(\bar{X}') = i + 1$, and $f(\bar{X}) \neq f(\bar{X}')$. Let $\hat{k} = \max\{k_1, N - k_2\}$.

THEOREM 2.8. *For any symmetric function f, $D_{N,Q}(f) = \Theta(N \log(\hat{k}/Q))$. In particular, $D_{N,Q}(\mathrm{PARITY}) = \Theta(N \log(N/Q))$.*

For the proof of this theorem see the end of §3.

3. Comparison trees. This section concerns noisy comparison trees. Our first claim is that binary searching and insertion in a balanced search tree does not require a blowup in noisy tree depth that grows with N. This result can be derived by modifying the algorithms of [23] or [25] and adapting them to our model, or from [14]. We present a different algorithm, which has the advantage that the ideas it is based on can also be used for other problems, where the techniques of [23] or [25] do not seem to apply (see Theorem 4.2). The algorithm is obtained by thinking of a noisy binary search as a *random walk* on the (exact) binary search tree.

In discussing upper bounds for searching among a set of elements $x_1 \leq x_2 \leq \cdots \leq x_N$ in a binary search tree, we will refer to our noisy comparison tree as an "algorithm" (rather than tree) to avoid confusion with the binary search tree. For simplifying the description we shall assume that the key being searched for is not in the tree (so that its insertion location has to be determined).

Each node of the tree represents a subinterval of $(-\infty, \infty]$, and is labeled by a pair representing the endpoints of this interval. In particular, each leaf of the search tree represents an interval between two consecutive input values. There are $N + 1$ leaves, with the ith $(1 \leq i \leq N + 1)$ representing $(x_{i-1}, x_i]$ (assume $x_0 = -\infty$ and $x_{N+1} = \infty$). For an internal node u of the tree, let T_u denote the subtree rooted at u. Then the intervals associated with the leaves of T_u are contiguous, and u represents the interval obtained by merging them. That is, u is labeled with the interval $(x_\ell, x_h]$, for $0 \leq \ell \leq h \leq N + 1$, where x_ℓ is the smallest endpoint of an interval associated with a leaf in T_u, and x_h is the largest such endpoint. The tree is nearly balanced, in the sense that for a vertex u labeled by $(x_\ell, x_h]$, the left child of u is labeled $(x_\ell, x_z]$ and the right child is labeled $(x_z, x_h]$, where $z = \lceil \frac{\ell+h}{2} \rceil$. The tree has depth $\lceil \log N \rceil$.

To search with unreliable comparisons we extend the tree in the following way: each leaf x_ℓ is a parent of a chain of length $m' = O(\log(N/Q))$. The nodes of the chain are labeled with the same interval as the leaf. (In practice, these chains can be implemented by counters representing the "depth" from the leaf.) Fig. 1 depicts the resulting tree for three values, $(x_1, x_2, x_3) = (2, 5, 7)$.

Let X (given, say, as x_{-1} in the input set) be the key being searched for in the tree. The search begins at the root of the tree, and advances or backtracks according to the results of the

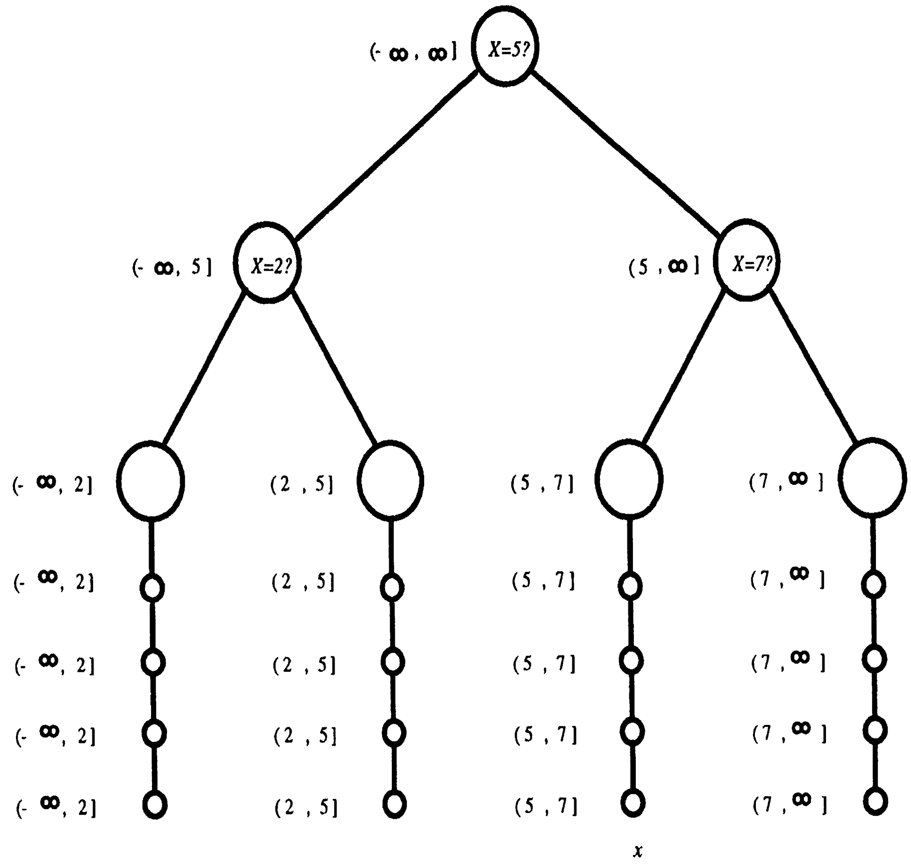


FIG. 1. *The extended comparison tree corresponding to the input list* (2, 5, 7).

comparisons. Whenever reaching a node u, the algorithm first checks that X really belongs to the interval $(x_\ell, x_h]$ associated with u, by comparing it to the endpoints of the interval. This test may either succeed, i.e., respond in $X > x_\ell$ and $X < x_h$, or fail, i.e., respond in $X < x_\ell$ or $X > x_h$ (or both). Such failure of the test may be due to noisy comparisons. However, the search algorithm always interprets a failure as revealing an inconsistency due to an earlier mistake, and consequently, the computation backtracks to the parent of u in the tree. If the test succeeds, on the other hand, then the computation proceeds to the appropriate child of u. That is, if u has two children, the algorithm compares X to x_z, the "central element" in u's interval (i.e., such that $z = \lceil \frac{\ell+h}{2} \rceil$), and continues accordingly.

The search is continued for $m = O(\log(N/Q))$ steps, $m < m'$ (hence it never reaches the endpoint of any chain). The outcome of the algorithm is the left endpoint of the interval labeling the node at which the search ends. For example, in the search tree depicted in Fig. 1, the search for the value $X = 6$ should terminate at the leaf marked x.

LEMMA 3.1. *For every $Q < 1/2$, the algorithm computes the correct location of X with probability at least $1 - Q$ in $O(\log(N/Q))$ steps.*

Proof. We model the search as a Markov process. Consider a leaf w of the extended tree T, and suppose that X belongs to the interval labeling this leaf. Orient all the edges of T towards w. Note that for every node v, exactly one adjacent edge is directed away from v and the other adjacent edges are directed towards v. Without loss of generality we can assume that the transition probability along the outgoing edge is at least $2/3$, and the probability of transitions along all other (incoming) edges is at most $1/3$. Otherwise, we can bootstrap the probability to $2/3$ by repeating each comparison $O(1)$ times and taking the majority.

Let m_f be a random variable counting the number of forward transitions (i.e., transitions in the direction of the edges) and let m_b denote the number of backward transitions ($m_f + m_b = m$). We need to show that $m_f - m_b > \log N$ with probability at least $1 - Q$, implying that the appropriate chain is reached. This follows from Chernoff's bound [4] for $m = c \log(N/Q)$, for a suitably chosen constant c. □

Using N insertions of the above algorithm, each with failure probability Q/N, yields a noisy comparison tree of depth $O(N \log(N/Q))$ for sorting.

THEOREM 3.2. (1) $D^{\text{Det}}_{N,Q}(\text{BINARY SEARCH}) = O(\log(N/Q))$.

(2) $D^{\text{Det}}_{N,Q}(\text{SORTING}) = O(N \log(N/Q))$.

(3) $D^{\text{Det}}_{N,Q}(\text{MERGING}) = O(N \log(N/Q))$.

We now present a noisy comparison tree of depth $O(N \log(m/Q)), m = \min\{K, N - K\}$ for selecting the Kth largest of N elements (in fact, the tree described can find *all* K largest elements, or all $N - K$ smallest elements for $K < N/2$). By symmetry, we need only consider the case $K < N/2$. Furthermore, the case $\sqrt{N} \leq K \leq N/2$ can be handled using our $O(N \log(N/Q))$ sorting algorithm. Thus we assume that $K < \sqrt{N}$. The idea in finding the Kth largest element when K is "small" is to use "tree selection" or "heapsort" (see Knuth, pp. 142–145 [12]). In essence, the algorithm operates as follows. Once a heap is created, the largest element can be extracted from the top of the heap, and "reheapifying" the rest of the elements requires at most $\log N$ noiseless comparisons. Thus, extracting the K largest elements can be done in $K \log N$ noiseless comparisons. By repeating each of these $K \log N$ comparisons $O(\log((K \log N)/Q))$ times in the face of noise we can extract each of the K largest elements from the heap with error probability at most $Q/2K$. Thus with $O(K \log N \log((K \log N)/Q))$ noisy comparisons we can extract the K largest elements with probability at most $Q/2$. For $K < \sqrt{N}$, this number of comparisons is $O(N \log(K/Q))$.

The only remaining problem is that of constructing the initial heap. In order to do this, run a "tournament" algorithm similar to the "NBA" algorithm in the introduction for finding the maximum with failure probability $Q/2K$. The algorithm takes $O(N \log(K/Q))$ steps, and each of the K largest elements has probability at most $Q/2K$ of being eliminated by a smaller element. Thus, with probability $1 - Q/2$, the initial heap is consistent with respect to the K largest elements, and this suffices for our purposes. Therefore we have the following theorem.

THEOREM 3.3. $D^{\text{Det}}_{N,Q}(K\text{-SEL}) = O(N \log(m/Q))$, *where* $m = \min\{K, N - K\}$.

In §2 we proved lower bounds on the threshold function in the noisy Boolean decision tree model, whereas in this section we prove upper bounds on selection in the noisy comparison tree model. We can use a reduction between the two problems to show that both bounds are tight (up to constant factors). But first, since the results are proven in different computational models, we need to show a reduction from the Boolean decision tree model to the comparison model.

LEMMA 3.4. *A noisy comparison between two Boolean variables can be implemented by a constant number of noisy queries.*

Proof. Query each of the two variables a constant number of times, obtain an estimate for each of the variables by taking the majority of the corresponding responses, and compare the estimates. □

For Boolean inputs, selecting the Kth largest element and testing (by $O(\log N)$ queries) if its value is 1, is equivalent to computing TH^N_K. The upper bound in Theorem 2.7 follows trivially. The upper bound for computing any symmetric function (Theorem 2.8) follows from the fact that the comparison tree for K-SEL actually finds *all* K largest (or $N-K$ smallest, if $K > N/2$) elements. By repeating K-SEL once with $K = k_1$ and once with $K = k_2$ (see the prologue to Theorem 2.8 for the interpretation of these parameters), and then querying each of the k_1 largest and each of the $N - k_2$ smallest elements $O(\log N)$ times, the value of any symmetric function can be established.

We now turn to lower bounds for the problems discussed above.

THEOREM 3.5. (1) $D^{\text{Prob}}_{N,Q}(\text{BINARY SEARCH}) = \Omega(\log(N/Q))$.

(2) $D^{\text{Prob}}_{N,Q}(\text{SORTING}) = \Omega(N \log(N/Q))$.

(3) $D^{\text{Prob}}_{N,Q}(K\text{-SEL}) = \Omega(N \log(m/Q))$, *where* $m = \min\{K, N-K\}$.

(4) $D^{\text{Prob}}_{N,Q}(\text{MERGING}) = \Omega(N \log(N/Q))$.

Proof. It is immediate that our searching and sorting algorithms are asymptotically optimal in the comparison model, hence claims (1) and (2).

Next, the fact that a comparison tree for K-SEL implies a comparison tree for TH^N_K enables us to derive claim (3) from Theorem 2.2.

Finally, a lower bound for MERGING (claim (4)) can be derived by a reduction from PARITY. We first show how a merging algorithm can be used to establish parity. Consider a vector $\bar{X} = (x_1, \dots, x_N)$ of Boolean inputs whose parity is to be established. Transform it to a vector of increasing integers $\bar{I} = (I_1, \dots, I_N)$, where for each j, $I_j = 2x_j + 3j$. Consider the merge operation of $\bar{I}$ with the vector $\bar{Y} = (Y_1, \dots, Y_N)$, where $Y_j = 3j + 1$. The result establishes the value of each of the x_j, since $I_j < Y_j$ iff $x_j = 0$. So in order to compute the parity of $\bar{X}$, it is sufficient to simulate the merging of $\bar{I}$ and $\bar{Y}$. Claim (4) now follows from Theorem 2.8 and the argument of Lemma 3.4. □

4. Parallel tournaments.

4. Parallel tournaments. In this section and the next we consider two problems on noisy N-processor PRAMs in which each comparison operation between two elements independently gives the correct result with probability at least p. In this section we discuss the problem of finding the maximum of N elements. Our solution can be implemented on an EREW parallel decision tree with at most $N/2$ comparisons per round in $O(\log(N/Q))$ rounds. Furthermore, each input element is involved in at most one comparison per round, and no element is ever copied to create a replica of the element. Because of its sporting interpretation, we will describe the algorithm in the tournament setting introduced in the introduction. Let us now describe this setting in more detail.

A parallel algorithm for computing the maximum is called a *tournament* if in each parallel step of the algorithm, each input element is involved in at most one comparison. A tournament is *deterministic* if the comparisons made at each step are uniquely determined by the results of comparisons in previous steps (no randomization is allowed). The *depth* of a tournament is the total number of parallel steps it takes. The *size* of a tournament is the total number of comparisons it involves. A tournament is *noisy* if comparisons might output the wrong answer. We consider noisy tournaments with a dynamic adversary. A noisy tournament is *Q-tolerant* if it outputs the maximal element with probability at least $1 - Q$.

THEOREM 4.1. *Any deterministic Q-tolerant tournament has depth* $\Omega(\log(N/Q))$ *and size* $\Omega(N \log(1/Q))$.

Proof. Let T be any Q-tolerant tournament. Let d denote its depth and s its size. Any Q-tolerant tournament is also a deterministic noise-free tournament for finding the maximum, hence its depth is at least $\log N$. Thus for $Q \geq N^{-1}$ we immediately derive that $d \geq \frac{\log(N/Q)}{2}$.

Assume now that $Q < N^{-1}$. For simplicity, we describe the argument as if a dynamic adversary were controlling the probability of error for each comparison. Fix an arbitrary input with a unique largest element. The adversary decides to introduce no noise in the comparisons. The tournament must output the correct maximal element. Now switch the indices of the largest and second largest elements in the input. Now the adversary introduces noise only in comparisons between the two largest elements, and T proceeds exactly as in the case that the inputs are not switched. Since there are at most d comparisons between the two largest elements, the probability that the algorithm returns the same index as that of the maximum element on both runs is $(1-p)^d$, implying that $d \geq (\log(1/Q))/(\log \frac{1}{1-p})$.

The bound on the size follows from Theorem 2.1, together with the equivalence of the models from Lemma 3.4. □

We remark that a stronger version of the above theorem, in which the algorithm is probabilistic and the adversary is static, can be proved along similar lines as those of Theorem 2.1.

We state an inequality, due to Hoeffding [8], to be used in the proof of the next theorem. Let X_i, for $1 \leq i \leq n$, be n independent random variables with identical probability distributions, each ranging over the interval $[a, b]$. Let $\bar{X}$ be a random variable denoting the average of the X_i's. Then

$$\text{Prob}(|\bar{X} - E(\bar{X})| \geq \delta) \leq 2e^{-\frac{2n\delta^2}{b-a}}.$$

THEOREM 4.2. *For every* $0 < Q < 1/2$ *there is a* Q*-tolerant deterministic tournament for finding the maximum with depth* $O(\log(N/Q))$ *and size* $O(N\log(1/Q))$ *simultaneously.*

The tournament we construct is similar in spirit to the noisy binary search procedure of §3. For simplicity (and without loss of generality) we assume that $N = 2^m - 1$ for some m. Create a balanced binary tree of depth m, and arbitrarily place one input element in each node (including leaves, root and internal nodes). The algorithm proceeds in rounds. In each round, many mini-tournaments are performed in parallel. Each mini-tournament involves three players, and the largest of the three wins with probability at least q, for some constant q to be computed later. The mini-tournaments are organized by partitioning the nodes of the tree into triplets in a way to be described shortly, and forming a mini-tournament between the three elements stored in each triplet. The partition into triplets depends on the round. In even rounds, each triplet consists of a node at an even level of the tree and its two children. Analogously, in odd rounds, each triplet consists of a node at an odd level and its two children. At the end of the round, the winner of each mini-tournament is stored at the parent node, and the two other elements are placed arbitrarily at the children. The whole procedure is repeated for $O(\log(N/Q))$ rounds.

We give some intuition on why our construction computes the maximum. The tournament is best described as a random walk taken by the maximal element, M, over the balanced binary tree. A win at a single mini-tournament may or may not advance M towards the root, depending on whether M is already placed at the parent node before the mini-tournament begins. But wins in two successive mini-tournaments advance M by at least one step. Likewise, if it loses one of two successive mini-tournaments, it may move away from the root by one step, and if it loses two successive mini-tournaments, it may move away from the root by two steps. Summing up the probabilities of these events, it follows that on the average, in two successive rounds, M is expected to decrease its distance to the root by at least $q^2 + 2q - 2$ steps. For $q > 15/16$, this value is greater than $3/4$, and so any ℓ rounds are expected to advance M by $3\ell/8$, and in less than $8m/3$ steps M is expected to reach the root. (Note that guaranteeing that M wins each mini-tournament with probability $q > 15/16$ can be achieved in a constant number of comparisons, since a mini-tournament involves only three players.)

Two parts are still missing from the construction. One is a method of preventing M from leaving the root once it reaches it. The other is a method of decreasing the total number of comparisons from $O(N\log(N/Q))$ to $O(N\log(1/Q))$. This is significant if $Q > N^{-c}$ asymptotically for any constant c.

In order to secure M at the root with high probability we adopt the following policy: an element stays at the root as long as it has won the majority of mini-tournaments since it last reached the root. We employ a root counter which is initialized to 0. In mini-tournaments which involve the root, if the element placed at the root wins the mini-tournament, the root counter is incremented by 1. If a different element wins, and the root counter is at 0, this element exchanges places with the root element. If the root element does not win and the root counter has value greater than 0, then the root counter is decremented by 1, and no exchange takes place.

LEMMA 4.3. *The probability that M is at the root after $d = 256\log(N/Q)$ rounds is at least* $1 - Q/2$.

Proof. Assume that some other element W wins the tournament, i.e., occupies the root by the end of the process. We do not decrease the probability of W ending up at the root if we let it begin the tournament placed at the root, and let it win without competition any mini-tournament in which M is not involved. This implies that during the whole tournament, only two elements, M and W, could have occupied the root. Furthermore, W played exactly $d/2$ mini-tournaments involving the root, losing at most $d/4$.

Now consider M's performance in the tournament. Envision a *scoring system*, where M starts with 0 *points*. Partition the rounds of the tournament into successive pairs of mini-tournaments. For each such pair, M's score is decremented by 1 point for each mini-tournament that it loses, and if M did not lose in any mini-tournaments, then its score is incremented by 1 point. In d rounds, M's score is expected to be at least $3d/8$. Applying the Hoeffding inequality with $d/2$ (for the d selected above), we get that with probability $1 - Q/N$, M's scores are at least $5d/16$ points. At most $\log n$ of these points can be accounted for as steps taking M from a leaf to just below the root. The other $5d/16 - \log n$ points must have been "wasted" on decrementing W's root counter. For d as in the lemma, this value is greater than $d/4$, contradicting our assumption that W ends the tournament at the root. □

Though the depth of the above tournament is $O(\log(N/Q))$ as desired, its size is $O(N\log(N/Q))$, which is too large (for $1/Q = o(N)$). In order to diminish the total number of comparisons when $1/Q < N$, we execute the following truncation procedure during the first $(\log N)/3$ rounds. After $O(i\log(1/Q))$ rounds, we delete the ith level from the bottom of the competition tree. This has the effect of reducing the number of parallel mini-tournaments by a constant factor every $O(\log(1/Q))$ rounds, and thus reducing the size of the first $(\log N)/3$ rounds of the competition to $O(N\log(1/Q))$. Since for $1/Q < N$ the total number of rounds is $O(\log N)$, it follows that the size of the whole competition remains $O(N\log(1/Q))$.

LEMMA 4.4. *The probability that M is at a leaf of the truncated tree after* $16i(\log(1/Q)+2)$ *rounds is less than* $Q/2^{i+1}$.

Proof. We may assume that M starts at a leaf of the tree. Observe that $(\log N)/3$ rounds are insufficient for M to reach the root, and thus we can ignore the effect of the root counter. In $\ell = 16i(\log(1/Q)+2)$ rounds, M is expected to advance by at least $3\ell/8 = 6i(\log(1/Q)+2)$ steps. The probability it advanced less than i steps is as specified in the lemma, by the Hoeffding inequality. □

We now have all the ingredients to complete the proof of Theorem 4.2.

Proof of Theorem 4.2. From Lemma 4.4 it follows that the probability that the maximal element M is lost in the truncation process is less than $Q/2$. Thus the total probability that M does not win the tournament is at most Q, completing the proof of the theorem. □

5. Parallel sorting. The main result of this section is an N processors randomized $O(\log N)$ time noisy sorting algorithm. We first present the algorithm in an N-parallel decision tree model, and then modify it to an N-processor PRAM algorithm.

Our proof uses the following results of Assaf and Upfal [2].

THEOREM 5.1 [2]. *There is a constant α, such that for every constant $c > 1$ there is an $N\log N$ processor deterministic* EREW-PRAM *algorithm that sorts N elements in the noisy comparison model in $O(c\alpha\log N)$ parallel time with failure probability $Q \leq N^{-c}$.*

(The result in [2] is stronger, the sorting algorithm is nonadaptive and can be implemented as a network of comparators; however the PRAM version is sufficient for the proofs in this section.)

THEOREM 5.2. *There is a constant β, such that for any constant $c > 1$ there is a randomized, noisy, parallel comparison tree (N comparisons per node) of depth $c\beta \log N$ that sorts N numbers with error probability $Q \le N^{-c}$.*

Proof. The algorithm has three phases. In the first phase it chooses a random sample of $N/\log N$ elements and sorts them by running the algorithm of Theorem 5.1 $(c+2)\beta \log N$ steps. Since $(N/\log N)^{-(c+2)} \le N^{-(c+1)}$ for sufficiently large N, the probability that the first phase fails to sort the sample correctly is bounded by $1/N^{c+1}$.

The second phase of the algorithm partitions the N elements into $\ell = N/\log N$ sets, $S_1, \ldots, S_\ell$, such that with probability $1 - 1/N^{c+1}$ all elements in each set S_i are not smaller than the $(i-1)$st sample element and are not larger than the ith sample element (in the correct sorted order). To achieve this, we assign one processor to each element. The processor runs the noisy binary search algorithm of Theorem 3.2 for $O((c+3)\log N)$ steps. The probability that one search fails is at most $1/N^{c+2}$, so that the probability that any element is misplaced is at most $1/N^{c+1}$.

The third phase sorts the $O(N/\log N)$ sets. The probability that any set has more than $(c+2)\log^2 N$ elements is bounded by

$$N\left(1 - \frac{(c+2)\log^2 N}{N}\right)^{N/\log N} \le N^{-(c+1)}.$$

In what follows we assume that all sets have no more than $(c+2)\log^2 N$ elements. We sort the sets in parallel in $O(\log N)$ parallel steps, using any logarithmic parallel algorithm such as the AKS network [1] or [5], repeating each comparison $\log N/\log\log N$ times and taking the majority value.

The probability that the majority of $\log N/\log\log N$ comparisons does not give the correct answer is bounded (using Chernoff bound) by

$$\exp\left(-\frac{1}{3}p\frac{\log N}{\log\log N}\left(1 - \frac{1}{2p}\right)^2\right).$$

Since the sorting algorithm of each set uses $O(\log^2 N \log\log N)$ comparisons, the probability that a given set is not correctly sorted is bounded by $\exp(-\theta \frac{\log N}{\log\log N})$ for some constant $\theta > 0$. Thus, the probability that more than $N/\log^3 N$ sets are not correctly sorted is bounded by

$$\binom{N/\log N}{N/\log^3 N}\exp\left(-\theta\frac{\log N}{\log\log N}\frac{N}{\log^3 N}\right) \le N^{-(c+1)}.$$

By comparing each element $O(\log N)$ times (sequentially) to its two neighbors in the computed order we can identify, with probability $1 - N^{-(c+1)}$, all the sets that are not correctly sorted. Since with high probability the total number of elements in these $O(N/\log^3 N)$ sets is bounded by $O(N/\log N)$ we can assign $\log N$ processors to each element and sort all the sets correctly, with probability $1 - 1/N^{c+1}$, in $O(\log N)$ additional parallel steps, using again the algorithm of Theorem 5.1 Summing the run-time and the failure probabilities of the three phases we get that the correct sorted order is computed in $O(c \log N)$ time with probability $1 - 1/N^c$. □

THEOREM 5.3. *There is a constant γ, such that for any constant $c > 1$ there is an N processor randomized* CRCW-PRAM *algorithm that sorts N elements in the noisy comparison model in $c\gamma \log N$ parallel time with failure probability $Q \le N^{-c}$.*

Proof. The three phases of the previous algorithm are implemented on an N processor randomized CRCW-PRAM as follows.

Phase one: Each element chooses to participate in the sample with probability $2N/\log N$. With probability $1 - 1/N^{c+1}$ the sample has at least $N/\log N$ elements and no more than $3N/\log N$ elements. Using an $O(\log N)$-time prefix sum algorithm we copy the sample to a second array. The fault tolerant sorting network can be directly modified to a PRAM algorithm.

Phase two: The binary search can be done in parallel by the N processors on a CREW-PRAM. The main complication in implementing this phase is in placing the elements in the sets. We use the counting method of Reischuk [20] to count the number of elements in each set, and allocate them in $N/\log N$ arrays.

Phase three: The only complication in implementing this phase is in assigning $\log N$ processors to each of the elements in sets that need to be sorted again. When these sets are identified, the allocation can be done by a $O(\log N)$-time prefix-sum procedure. □

6. Extensions and open problems. Using reductions from the bounds given above, it is possible to derive tight bounds on the depths of noisy tree for the following problems: finding the leftmost 1, UNARY-BINARY, COMPARISON, ADDITION and MATCHING (see [3] for definitions).

The results of §2 can also be extended to show that there is a noisy Boolean decision tree of depth $O(N \log(1/Q))$ for any function that can be computed by a constant-depth formula of size N.

In Theorem 2.8 we characterized the noisy decision tree complexity of all symmetric functions. Obtaining such a characterization for general functions is a major open question. Some progress was achieved by Kenyon and King [10], who showed that $O(N \log(k/Q))$ queries suffice to compute any function f that can be represented either in k-DNF form or in k-CNF form. As for lower bounds, Reischuk and Schmeltz [21] showed that almost all functions require $\Theta(N \log(N/Q))$ queries. A simpler proof of this result is presented in [6].

An interesting open question is to give a deterministic noisy PRAM algorithm for sorting. We conjecture that there is no noisy sorting *network* of size $O(N \log N)$ that sorts N elements with polynomially small error probability.

Acknowledgments. We thank Noga Alon and Yossi Azar for helpful discussions, and for directing us to some of the references. Thanks are also due to Oded Goldreich and two anonymous referees for their illuminating comments on previous drafts of the paper.

REFERENCES

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

[2] S. Assaf and E. Upfal, *Fault tolerant sorting network*, in 31st Annual Symposium on Foundations of Computer Science, pp. 275–284, October 1990.

[3] A. K. Chandra, L. Stockmeyer, and U. Vishkin, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[4] H. Chernoff, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Annals of Math. Stat., 23 (1952), pp. 493–509.

[5] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[6] U. Feige, *On the complexity of finite random functions*, Inform. Process. Lett., 44 (1992), pp. 295–296.

[7] J. Hastad, F. T. Leighton, and M. Newman, *Reconfiguring a hypercube in the presence of faults*, in 19th Annual Symposium on Theory of Computing, pp. 274–284, 1987.

[8] W. Hoeffding, *Probability inequalities for sums of bounded random variables*, J. Amer. Stat. Assoc., 58 (1963), pp. 13–30.

[9] R. M. Karp, Personal communication, Berkeley, CA, 1989.

[10] C. Kenyon and V. King, *On Boolean decision trees with faulty nodes*, Proc. of the Israel Symposium on the Theory of Computing and Systems, 1992, Springer-Verlag, New York.

[11] C. Kenyon-Mathieu and A. C. Yao, *On evaluating Boolean functions with unrealiable tests*, Int. J. of Foundations of Computer Science, 1 (1990), pp. 1–10.

[12] D. E. Knuth, *Sorting and Searching, The Art of Computer Programming*, vol. 3. Addison-Wesley, Reading, MA, 1973.

[13] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. ACM, 27 (1980), pp. 228–234.

[14] A. Pelc, *Serching with known error probability*, Theoret. Comput. Sci., 63 (1989), pp. 185–202.

[15] ———, *Sorting with random errors*, Technical Report TR # RR 89/06-12, Univ. du Quebec a Hull, Quebec, Canada, 1989.

[16] N. Pippenger, *On networks of noisy gates*, in 26th Annual Symposium on Foundations of Computer Science, pp. 30–38, 1985.

[17] N. Pippenger, G. D. Stamoulis, and J. N. Tsitsiklis, *On a lower bound for the redundancy of reliable networks with noisy gates*, IEEE Transactions on Information Theory, to appear.

[18] B. Ravikumar, K. Ganesan, and K. B. Lakshmanan, *On selecting the largest element in spite of erroneous information*, in Proc. 4th Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., pp. 88–99, Springer-Verlag, New York, 1987.

[19] B. Ravikumar and K. B. Lakshmanan, *Coping with known patterns of lies in a search game*, Theoret. Comput. Sci., 33 (1984), pp. 85–94.

[20] R. Reischuk, *Probabilistic parallel algorithms for sorting and selection*, SIAM J. Comput., 14 (1985), pp. 396–409.

[21] R. Reischuk and B. Schmeltz, *Reliable computation with noisy circuits and decision trees – a general $n \log n$ lower bound*, in 32nd Annual Symposium on Foundations of Computer Science, pp. 602–611, San Juan, Puerto Rico, 1991.

[22] A. Renyi, *On a problem in information theory*, in Selected Papers of Alfred Renyi, volume 2, P. Turan, ed., pp. 631–638. Akademiai Kiado, Budapest, 1976.

[23] R. L. Rivest, A. R. Meyer, D. J. Kleitman, K. Winklmann, and J. Spencer, *Coping with errors in binary search procedures*, J. Comput. System Sciences, 20 (1980), pp. 396–404.

[24] M. Saks and A. Wigderson, *Probabilistic Boolean decision trees and the complexity of evaluating game trees*, in 27th Annual Symposium on Foundations of Computer Science, pp. 29–38, Toronto, Ontario, 1986.

[25] J. P. M. Schalkwijk, *A class of simple and optimal strategies for block coding on the binary symmetric channel with noiseless feedback*, IEEE Trans. Inform. Theory, 17 (1971), pp. 283–283.

[26] A. C. Yao and F. F. Yao, *On fault-tolerant networks for sorting*, SIAM J. Comput., 14 (1985), pp. 120–128.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 1019–1025, October 1994

009

ON COMPETITIVE GROUP TESTING*

DING-ZHU DU† AND HAESUN PARK†

Abstract. In many fault detection problems, the goal is to identify defective items from a set of items with a minimum number of tests. Each test is on a subset of items, which tells whether the subset contains a defective item or not. The concept of *competitive* algorithm has been developed to relate the properties of the group testing algorithms that assume that the number of defective items d is known, to those without any a priori knowledge on d. A new concept of *strongly competitive* algorithm is defined that relates different characteristics of these two classes of algorithms and present an interesting relationship between the two concepts *competitive* and *strongly competitive*. A strongly competitive algorithm is also presented.

Key words. competitive algorithm, group testing, fault detection, strongly competitive algorithm

AMS subject classifications. 68P10, 68Q25, 68R05

1. Introduction.

Consider a set of n items in which some items are *defective* and others are *good*. The problem is to identify them by a sequence of tests. Each test is on a subset of items, which tells us whether the subset contains a defective item or not. In the former case, the subset is said to be *contaminated* and in the latter case, the subset is said to be *pure*. The problem has applications in high speed computer networks [4], medical examination [1], [2], quantity searching [3], and statistics [1], etc. In the literature, the problem has been named *group testing*.

A classic model for group testing is to assume that the number of defective items is known. This assumption is restrictive since in practice this number is usually unknown a priori and it can be discovered only after testing. If the number of defective items is unknown at the beginning, how do we design and analyze algorithms? Motivated from the study of on-line problems [16], [17], Du and Hwang [9] proposed the concept of competitive algorithm for the group testing problem as follows.

For an algorithm α, let $N_\alpha(s \mid d, n)$ ($N_\alpha(s \mid n)$) be the number of tests that algorithm α spends on a sample s of n items under the condition that d, the number of defective items, is known (unknown). Denote

$$M_\alpha(d, n) = \max_{s \in \mathcal{A}(d,n)} N_\alpha(s \mid d, n),$$

$$M(d, n) = \min_{\alpha} M_\alpha(d, n),$$

$$M_\alpha(d \mid n) = \max_{s \in \mathcal{A}(d,n)} N_\alpha(s \mid n),$$

where $\mathcal{A}(d, n)$ is the set of samples of n items containing d defective items. An algorithm α is called a *c-competitive* algorithm if there exists a constant a such that for $0 < d < n$, $M_\alpha(d \mid n) \leq c \cdot M(d, n) + a$. Note that in the definition we exclude the case $d = n$ because $M(n, n) = 0$ and $M_\alpha(d \mid n)$ may depend on n. The exclusion of the case $d = 0$ is not necessary, but is convenient. We will remark on this in §4. A c-competitive algorithm for a constant c is simply called a competitive algorithm while c is called the *competitive ratio* of the algorithm.

*Received by the editors March 22, 1993; accepted for publication (in revised form) June 28, 1993.

†Computer Science Department, University of Minnesota, Minneapolis, Minnesota 55455. The work of the first author was supported in part by the National Science Foundation under grant CCR-9208913 and by the Institute of Applied Mathematics, Chinese Academy of Sciences, Beijing, China.

Du and Hwang [9] presented two competitive algorithms, a bisecting algorithm with competitive ratio 2.75 and a hybrid algorithm α that has the following property:

$$(*) \qquad \lim_{n\to\infty} \frac{M_\alpha(d \mid n)}{M(d,n)} = 1.$$

They conjectured that there exists a bisecting algorithm A such that

$$(1) \qquad M_A(d \mid n) \leq 2M(d,n) + 1 \quad \text{for } 0 \leq d \leq n-1.$$

Bar-Noy, Hwang, Kessler and Kutten [4] discovered a 2.15-competitive algorithm with a new technique, called a *doubling algorithm*. Du, Xue, Sun, and Cheng [12] introduced a new method to analyze the competitive algorithms. They proved that the bisecting algorithm appeared in an earlier version of Du and Hwang's paper [10] actually satisfies (1). Du and Hwang [11] pointed out that with a similar method, the doubling algorithm can be proved to have competitive ratio 2, too. Du, Xue, Sun, and Cheng [12] also obtained a competitive algorithm with ratio 1.65 by modifying the doubling algorithm. Neither the doubling algorithm nor its modified version has the property $(*)$.

In this paper, we propose a new competitiveness for group testing and prove that a competitive algorithm in the new sense must both satisfy property $(*)$ and be competitive in the original sense. We also present an algorithm which is competitive in the new sense.

2. Strong competitiveness. A number of papers [5]–[7], [14]–[15] have appeared on group testing algorithms for $1 \leq d \leq 3$. In those papers, the following number was studied:

$$n(d,k) = \max\{n \mid M(d,n) \leq k\}.$$

Motivated from previous work on this number, we defined the second competitiveness as follows.

Consider an algorithm α. Define $n_\alpha(d \mid k) = \max\{n \mid M_\alpha(d \mid n) \leq k\}$. An algorithm α is called a *strongly c-competitive* algorithm if there exists a constant a such that for every $d \geq 1$ and $k \geq 1$, $n(d,k) \leq c \cdot n_\alpha(d \mid k) + a$. Note that we have to exclude the case $d = 0$ because $n(0,k) = \infty$ for $k \geq 1$. A strongly c-competitive algorithm for a constant c is called a strongly competitive algorithm. An interesting property of strong competitiveness is as follows.

THEOREM 2.1. *Every strongly competitive algorithm α is a competitive algorithm satisfying the following condition*:

$$\lim_{n\to\infty} \frac{M_\alpha(d \mid n)}{M(d,n)} = 1.$$

Before proving this theorem, let us first state a lemma.

LEMMA 2.2. *For* $1 \leq d \leq n$,

$$d \log_2 \frac{n}{d} \leq M(d,n) \leq d \log_2 \frac{n}{d} + 4d.$$

Proof. There are $\binom{n}{d}$ samples. Each test divides these samples into two sets. So, the information lower bound for $M(d,n)$ is $\lceil \log_2 \binom{n}{d} \rceil$. (A better bound can be found in [8] and [13].) The first inequality follows immediately from $\binom{n}{d} \geq \left(\frac{n}{d}\right)^d$. The second inequality will be proved in the next section. □

Now we prove Theorem 2.1.

Proof of Theorem 2.1. Let $n(d, k) \le c \cdot n_\alpha(d \mid k) + a$. Without loss of generality, assume that c and a are positive integers; otherwise, we may use two positive integers bigger than c and a instead of c and a. From the above inequality, we obtain

$$\begin{aligned} M_\alpha(d \mid n) &= \min\{k \mid n_\alpha(d \mid k) \ge n\} \\ &\le \min\{k \mid (n(d,k) - a)/c \ge n\} = M(d, cn + a). \end{aligned}$$

Thus, for $d \ge 1$,

$$1 \le \frac{M_\alpha(d \mid n)}{M(d,n)} \le \frac{M(d, cn+a)}{M(d,n)} \le \frac{d \log_2 \frac{cn+a}{d} + 4d}{d \log_2 \frac{n}{d}}.$$

Clearly, the right side is bounded by a constant for $d < n$. Thus, α is a competitive algorithm. Moreover, the right side approaches one as n goes to infinity and d is fixed. Therefore $\lim_{n \to \infty} M_\alpha(d \mid n)/M(d,n) = 1$ for $d \ge 1$. □

The inverse of the above theorem is not true. Next, we give a sufficient condition for strong competitivity.

THEOREM 2.3. *If for algorithm α there exists a constant c such that for $1 \le d \le n - 1$,*

$$M_\alpha(d \mid n) \le d \log_2 \frac{n}{d} + cd,$$

then α is strongly competitive.

Proof. Note that

$$\begin{aligned} n_\alpha(d \mid k) &= \max\{n \mid M_\alpha(d \mid n) \le k\} \\ &\ge \max\left\{n \mid d \log_2 \frac{n}{d} + cd \le k\right\} \\ &\ge d2^{\frac{k}{d} - c} - 1 \end{aligned}$$

and

$$\begin{aligned} n(d, k) &= \max\{n \mid M(d,n) \le k\} \\ &\le \max\left\{n \mid d \log_2 \frac{n}{d} \le k\right\} \\ &\le d2^{\frac{k}{d}}. \end{aligned}$$

Thus,

$$n(d,k) \le 2^c(n_\alpha(d \mid k) + 1). \qquad \square$$

3. Strongly competitive algorithm. A bisecting algorithm for group testing can be simply described as follows: Starting from the input set of items, the algorithm tests a set and bisects it when the set is found to be contaminated. Continue to do this until every item is either in a pure set or in a contaminated singleton. A lot of tests are wasted in the bisecting algorithm in the following situation: When X and X' both are contaminated sets with $X' \subset X$, then the test on X' would render the test on X useless. From this observation, we make the following improvement: Once a contaminated set is found, a defective item is eliminated from the set. This idea may decrease the frequency of the occurrence of the above situation, which results in a strongly competitive algorithm.

First, we introduce a procedure ELIMINATE which eliminates a defective item from a contaminated set X with $\lceil \log_2 |X| \rceil$ tests. Note that performing a test on set X will be written as TEST(X). Let G and D be containers for good items and defective items, respectively.

```
Procedure ELIMINATE($X$);
    $Y := X$;
    while $|Y| \geq 2$ do begin
        $Y' := \lceil |Y|/2 \rceil$ items from $Y$;
        TEST($Y'$);
        if $Y'$ is contaminated then $Y := Y'$
            else $Y := Y \setminus Y'$;
                 $G := G \cup Y'$;
                 $X := X \setminus Y'$;
    end-while;
    $X := X \setminus Y$;
    $D := D \cup Y$;
end-procedure.
```

Now we give an algorithm as follows.

```
ALGORITHM A:
input $S$;
$G := \emptyset$;
$D := \emptyset$;
$Q := \{S\}$;
repeat
    pop $X$ from queue $Q$;
    TEST($X$);
    if $X$ is pure
    then $G := G \cup X$
    else ELIMINATE($X$);
        if $|X| = 1$ then push $X$ into queue $Q$;
        if $|X| \geq 2$ then bisect $X$ into $X'$ and $X''$ such that $|X'| = 2^{\lceil \log_2 |X| \rceil - 1}$ and
            $X'' = X \setminus X'$ and push $X'$ and $X''$ into queue $Q$;
until $Q = \emptyset$.
end-algorithm.
```

Next, we analyze Algorithm A. We first consider n which is a power of 2 since the analysis is easier in this case.

LEMMA 3.1. *Let n be a power of 2. Then for $0 \leq d \leq n$.*

$$M_A(d \mid n) \leq d \log_2 \frac{n}{d+1} + 4d - \log_2(d+1) + 1.$$

Proof. Let $n = 2^u$, $v = \lfloor \log_2(d+1) \rfloor$ and $v' = \log_2(d+1) - v$. Note that detecting a defective item from a set of size s by procedure ELIMINATE needs $\lceil \log_2 s \rceil$ TESTs. In Algorithm A, once a contaminated set is discovered, a defective item is eliminated from the set and then the remainder is divided into two sets if the remainder contains at least two items. It is not hard to see that procedure ELIMINATE is applied to at most one set of size n, at most two sets of sizes between $1 + n/4$ and $n/2, \ldots$, and in general, at most 2^i sets of sizes between $1 + n/2^{i+1}$ and $n/2^i$ for $i < v$. Thus, the number of TESTs required by procedure ELIMINATE is at most

$$\sum_{i=0}^{v-1} 2^i(u-i) + \left(d - \sum_{i=0}^{v-1} 2^i\right)(u-v)$$

$$= u(2^v - 1) - (v2^v - 2^{v+1} + 2) + (d - 2^v + 1)(u - v)$$

$$
\begin{aligned}
&= ud - v(d+1) + 2^{v+1} - 2\\
&= d(u-v-v') + v'd - v + 2^{v+1} - 2\\
&= d(u-v-v') + (v' + 2^{1-v'})(d+1) - 2 - \log_2(d+1)\\
&\le d\log_2 \frac{n}{d+1} + 2d - \log_2(d+1).
\end{aligned}
$$

The last inequality holds because $v' + 2^{1-v'} \le 2$ for $0 \le v' < 1$. In fact, $v' + 2^{1-v'}$ is a convex function with value 2 at $v' = 0$ and $v' = 1$.

Now, consider the tree T^* which is built up by the bisecting process as follows: The node set of T^* consists of S and all sets X' and X'' appearing in the computation. A node X is the father of two sons X' and X'' if and only if X' and X'' are obtained by bisecting X after eliminating a defective item. Clearly, every internal node is a contaminated set from which a defective item is eliminated. Thus, T^* has at most d internal nodes. It follows that the total number of nodes of T^* is at most $2d+1$. Therefore,

$$M_A(d \mid n) \le d\log_2 \frac{n}{d+1} + 4d - \log_2(d+1) + 1. \qquad \square$$

For convenience, we assume that the value of function $d\log_2 \frac{n}{d}$ at $d = 0$ is 0 because $\lim_{d\to 0} d\log_2 \frac{n}{d} = 0$. The following lemma is an important tool for our analysis.

LEMMA 3.2. *Let $d = d_1 + d_2$ and $n = n_1 + n_2$, where $d_1 \ge 0, d_2 \ge 0, n_1 > 0$ and $n_2 > 0$. Then*

$$d_1 \log_2 \frac{n_1}{d_1} + d_2 \log_2 \frac{n_2}{d_2} \le d \log_2 \frac{n}{d}.$$

Proof. Note that $d^2/dx^2(x\log_2 \frac{1}{x}) = -1/(x\ln 2) < 0$ for $x > 0$, so $-x\log_2 x$ is a concave function. Thus,

$$
\begin{aligned}
&d_1 \log_2 \frac{n_1}{d_1} + d_2 \log_2 \frac{n_2}{d_2}\\
&= n\left(\frac{n_1}{n}\frac{d_1}{n_1}\log_2 \frac{n_1}{d_1} + \frac{n_2}{n}\frac{d_2}{n_2}\log_2 \frac{n_2}{d_2}\right)\\
&\le n\left(\frac{d}{n}\log_2 \frac{n}{d}\right)\\
&= d\log_2 \frac{n}{d}. \qquad \square
\end{aligned}
$$

LEMMA 3.3. *For $1 \le d \le n$,*

$$M_A(d \mid n) \le d\left(\log_2 \frac{n}{d} + 4\right).$$

Proof. We prove it by induction on d. For $d = 1$, the algorithm finds the only defective item with $\lceil \log_2 n \rceil + 1$ TESTs which is clearly bounded by $\log_2 n + 4$. For $d > 1$, the algorithm eliminates the first defective item with $\lceil \log_2 n \rceil + 1$ TESTs, and bisects the remaining $n-1$ items into two sets S' and S'' of size n_1 and n_2, respectively, where $n_1 = 2^{u-1}$ with $u = \lceil \log_2(n-1) \rceil$. Suppose that S' and S'' contain d_1 and d_2 defective items, respectively. So, $d_1 + d_2 + 1 = d$ and $n_1 + n_2 + 1 = n$. Then by Lemma 3.1, the number of TESTs for identifying items in S' is at most

$$d_1\left(\log_2 \frac{n_1}{d_1+1} + 4\right) - \log_2(d_1+1) + 1.$$

Next, we consider two cases.

Case 1. $d_2 > 0$. By the induction hypothesis, all defective items in S'' can be identified in at most $d_2(\log_2(n_2/d_2) + 4)$ tests. Thus, the total number of TESTs is at most

$$\begin{aligned}
&(\lceil \log_2 n \rceil + 1) + \left(d_1 \log_2 \frac{n_1}{d_1+1} + 4d_1 - \log_1(d_1+1) + 1\right) + \left(d_2 \log_2 \frac{n_2}{d_2} + 4d_2\right)\\
&\leq u + 3 + d_1\left(\log_2 \frac{n_1}{d_1+1} + 4\right) - \log_2(d_1+1) + d_2\left(\log_2 \frac{n_2}{d_2} + 4\right)\\
&= (d_1+1)\left(\log_2 \frac{n_1}{d_1+1} + 4\right) + d_2\left(\log_2 \frac{n_2}{d_2} + 4\right) \qquad \text{(since } u = 1 + \log_2 n_1\text{)}\\
&\leq d\left(\log_2 \frac{n-1}{d} + 4\right) \qquad \text{(by Lemma 3.2)}\\
&< d\left(\log_2 \frac{n}{d} + 4\right).
\end{aligned}$$

Case 2. $n \neq 2^u + 1$ and $d_2 = 0$. In this case, $u = \lceil \log_2(n-1) \rceil = \lceil \log_2 n \rceil$ and the algorithm uses one TEST to detect S''. Thus, the total number of TESTs is at most

$$\begin{aligned}
&(\lceil \log_2 n \rceil + 1) + \left(d_1 \log_2 \frac{n_1}{d_1+1} + 4d_1 - \log_1(d_1+1) + 1\right) + 1\\
&\leq u + 3 + d_1\left(\log_2 \frac{n_1}{d_1+1} + 4\right) - \log_2(d_1+1)\\
&= (d_1+1)\left(\log_2 \frac{n_1}{d_1+1} + 4\right) \qquad \text{(since } u = 1 + \log_2 n_1\text{)}\\
&< d\left(\log_2 \frac{n}{d} + 3\right).
\end{aligned}$$

Case 3. $n = 2^u + 1$ and $d_2 = 0$. In this case, $u + 1 = \lceil \log_2 n \rceil$ and $n_1 = n_2 = 2^{u-1}$ and the total number of TESTs is at most

$$\begin{aligned}
&u + 4 + d_1\left(\log_2 \frac{n_1}{d_1+1} + 4\right) - \log_2(d_1+1)\\
&\leq 1 + d\left(\log_2 \frac{n_1}{d} + 4\right)\\
&\leq d\left(\log_2 \frac{n}{d} + 4\right). \qquad \square
\end{aligned}$$

It is an immediate consequence that $M(d, n) \leq d \log_2 \frac{n}{d} + 4d$ for $1 \leq d \leq n$.

THEOREM 3.4. *Algorithm* A *is strongly competitive.*

Proof. It follows immediately from Lemma 3.3 and Theorem 2.3. □

4. Discussions.

In the definition of competitive algorithm, the exclusion of the case $d = 0$ is not necessary because all current existing competitive algorithms spend a constant number of tests in this case. However, this case makes some unnecessary noise. In fact, if we include this case in the definition, all results in this paper still hold provided we add "$M_\alpha(0, n)$ is a constant" in the definition of strongly competitive and in the assumption of Theorem 2.3 and note that $M_A(0, n) = 1$ for Algorithm A in §3.

The strongly competitive algorithms are near optimal when $d << n$. This happens often in practice.

The hybrid algorithm in [9] is also a strongly competitive algorithm. However, the competitive ratio $2^{5.28}$ is bigger than that of the algorithm which we presented in the last section. We can design other strongly competitive algorithms with the same upper bound as that in Lemma 3.3. The coefficient 4 for the term d in the upper bound seems to be a barrier.

The lower bound for $M(d, n)$ has been established by [8], [13]. Whether the ratio $(M(d \mid n) - M(d, n))/d$ is close to 4 or close to 1 is an open question.

REFERENCES

[1] R. Ahlswede and I. Wegener, *Search Problems*, John Wiley and Sons, New York, 1987.

[2] M. Aigner, *Combinatorial Search*, John Wiley and Sons, New York, 1988.

[3] J. A. Aslam and A. Dhagat, *Searching in the presence of linearly bounded errors*, Proceedings of 23rd ACM Symp. on Theory of Computing, New Orleans, Louisiana, 1991, pp. 486–493.

[4] A. Bar-Noy, F. K. Hwang, I. Kessler, and S. Kutten, *A new competitive algorithm for group testing*, Discrete Applied Mathematics, to appear.

[5] G. J. Chang and F. K. Hwang, *A group testing problem on two disjoint sets*, SIAM J. Algebraic Discrete Meth., 2 (1981), pp. 35–38.

[6] G. J. Chang, F. K. Hwang, and S. Lin, *Group testing with two defectives*, Discrete Appl. Math., 4 (1982), pp. 97–102.

[7] X. F. Chang, F. K. Hwang, and J. Weng, *Optimal detection of two defectives with a parity check device*, SIAM J. Discrete Math., 1 (1988), pp. 38–44.

[8] D. Z. Du and F. K. Hwang, *Minimizing a combinatorial function*, SIAM J. Algebraic Discrete Meth., 3 (1982), pp. 523-528.

[9] ———, *Competitive group testing*, in L.A. McGeoch and D.D. Sleator, eds., Proceedings of Workshop on Online Algorithms, Feb. 1991, pp. 125–134.

[10] ———, *Competitive group testing*, DIMACS Technical Report 90-62, New Brunswick, NJ, 1990.

[11] ———, *Combinatorial Group Testing and Its Applications*, World Scientific Co., Singapore, 1993.

[12] D. Z. Du, G.-L. Xue, S.-Z. Sun, and S.-W. Cheng, *Modifications of competitive group testing*, SIAM J. Computing, 23 (1994), pp. 82–96.

[13] M. C. Hu, F. K. Hwang and J. K. Wang, *A boundary problem for group testing*, SIAM J. Algebraic Discrete Meth., 2 (1981), pp. 81–87.

[14] F. K. Hwang, *A tale of two coins*, Amer. Math. Monthly, 84 (1987), pp. 121–129.

[15] K.-I. Ko, *Searching for two objects by underweight feedback*, SIAM J. Discrete Math., 1 (1988), pp. 65–70.

[16] M. S. Manasse, L. A. McGeoch, and D. D. Sleator, *Competitive algorithms for on-line problems*, Proceedings of 20th ACM Symp. on Theory of Computing, Chicago, Illinois, 1988, pp. 322–333.

[17] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of ACM*, 28 (1985), pp. 202–208.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 1026–1049, October 1994

A UNIFORM CIRCUIT LOWER BOUND FOR THE PERMANENT*

ERIC ALLENDER† AND VIVEK GORE‡

Abstract. The authors show that uniform families of ACC circuits of subexponential size cannot compute the permanent function. This also implies similar lower bounds for certain sets in PP. This is one of the very few examples of a lower bound in circuit complexity whose proof hinges on the uniformity condition; it is still unknown if there is any set in Ntime ($2^{n^{O(1)}}$) that does not have nonuniform ACC circuits.

Key words. circuit complexity, uniformity, permanent, lower bounds, complexity classes

AMS subject classifications. 68Q05, 68Q15, 03D15

1. Introduction.

Circuit complexity classes consisting of circuits of constant depth and polynomial size have been intensely studied in the last decade. The first such class to be studied was AC^0, the class of languages accepted by polynomial size, constant depth circuits consisting of NOT gates and unbounded fan-in AND and OR gates. Machinery for proving lower bounds for AC^0 has been developed in a series of papers, culminating in the powerful and elegant techniques of [18], [29], [3]. These papers provide exponential size lower bounds for constant depth circuits computing the PARITY function. These lower bounds prompted people to look at constant depth, polynomial size circuits with PARITY gates along with AND, OR, and NOT gates, but Razborov [22] proved that these circuits could not compute the MAJORITY function. Smolensky [25] extended Razborov's method to show that an AC^0 circuit with MOD_p gates cannot compute the MOD_q function if p and q are distinct primes. This implies that no AC^0 circuit containing MOD gates for a single prime can compute the MAJORITY function. Therefore, the next natural extension of the above class was to allow MOD_m gates for composite moduli m. This extension is known as the class ACC, and it was introduced (implicitly) by Barrington in [4]. Although there has been a fair amount of research on ACC, we still do not know much about this class except the trivial fact that $AC^0 \subsetneq ACC \subseteq NC^1$ where NC^1 is the class of languages accepted by polynomial size, $O(\log n)$ depth circuits with NOT gates, and bounded fan-in AND and OR gates. Barrington [4] has conjectured that $ACC \subsetneq NC^1$.

Yao [30] proved the first nontrivial upper bounds on the power of ACC circuits, showing that each set in ACC is accepted by a family of depth three threshold circuits of size $2^{(\log n)^{O(1)}}$; these bounds were slightly improved by Beigel and Tarui [10]. These results have been proved for nonuniform ACC. We are, however, interested in the uniform version of ACC.

A circuit family consists of a sequence of circuits $C_1, C_2, \ldots,$ where circuit C_n takes n Boolean inputs. The circuit family is *uniform* if a description of C_n can be computed efficiently from n; otherwise the circuit family is said to be *nonuniform*. The original motivation for studying uniform circuit families came from a desire to relate time and space complexity classes to circuit complexity (see, e.g., [11]). Some sort of uniformity condition is essential for this endeavor to succeed, since it is an easy observation that there are sets with trivial circuit complexity that are not even recursive. The question of exactly which uniformity condition one should use has proved to be somewhat controversial, and largely it has been a matter of taste.

*Received by the editors July 9, 1992; accepted for publication (in revised form) June 16, 1993.

†Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903 (allender@cs.rutgers.edu). The work of this author was supported in part by National Science Foundation grants CCR-9000045 and CCR-9204874, and was conducted in part while visiting Princeton University, Princeton, New Jersey 08544.

‡Program in Computing, Department of Mathematics, University of California, Los Angeles, California 90024-1555 (gore@math.ucla.edu). This work was conducted while the author was a student at Rutgers University, and was supported in part by a Rutgers University Graduate Excellence Fellowship, and by National Science Foundation grant CCR-9204874.

When providing upper bounds, or when defining complexity classes, as a practical matter it usually makes no difference which uniformity condition one uses. For example, Ruzzo [23] considers a number of related uniformity conditions, and shows that, for all $k \geq 2$, NC^k consists of languages defined by uniform circuits of polynomial size and $O((\log n)^k)$ depth, no matter which of those uniformity conditions is considered. For very small complexity classes, however, the uniformity condition is sometimes crucial. For example, P-uniform NC^1 circuits are known for division [8], but it remains an open question whether one can improve this result using a more restrictive uniformity condition. Similarly, [6] presents a number of beautiful characterizations of subclasses of NC^1 using Dlogtime uniformity, but these characterizations are not believed to hold if less restrictive uniformity conditions are used. In this paper, we consider uniform circuits out of necessity. The lower bounds that we present are not known to hold in the nonuniform setting.

Before we can state our results, we need a few technical definitions. We are interested in two classes of subexponential functions that we call subexp and subsubexp. Let us call a function f *constructible* if $f(n) = 2^{g(n)}$, where $g(n)$ can be computed from n (in binary) in time polynomial in $g(n)$. Let subexp denote the class of all monotonic functions that are bounded above by some constructible function f such that $\forall \epsilon > 0$, $f(n) = o(2^{n^\epsilon})$. Let subsubexp denote any class of monotonic functions closed under composition with polynomials, such that for any two functions f and g in this class, the composition of f and g is in subexp.

A typical example of a function in subexp is $2^{n^{1/\log^* n}}$, and typical choices for subsubexp are $n^{\log^{o(1)} n}$ or $2^{(\log n)^{O(\log\log n)}}$. It is not hard to prove that if s is in subexp, then so is $s^{(\log s)^k}$, for any constant k.

In this paper, we provide lower bounds for the classes of languages accepted by uniform circuit families of ACC circuits of subsubexponential and subexponential size. Let those classes be denoted by ACC(subexp), and ACC(subsubexp). Formal definitions can be found in §2 (Definition 2.10). For the rest of this section, we assume that ACC, ACC(subexp), and ACC(subsubexp), denote the uniform versions of these classes for the notion of uniformity defined in §2 (Definition 2.9). Any other notions of uniformity that we use will be mentioned explicitly. We show that PERM (the permanent of a matrix) is not in ACC(subexp), and that PP $\not\subseteq$ ACC(subsubexp). We are also able to show that ACC $\subsetneq$ $\mathrm{C_=P}$ and that $\mathrm{C_=P}$ $\not\subseteq$ ACC(subsubexp). Our main tool in proving these results is the following theorem.

THEOREM 1.1. *There is a set Y in* PP *such that* ACC(subexp), $\subseteq \mathrm{Dtime}(n^2)^Y$.

Theorem 1.1 trivially gives us an important corollary (which also follows from a more general lower bound proved in Theorem 3.5 later in the paper).

COROLLARY 1.2. ACC $\subsetneq$ PP.

Proof. Theorem 1.1 implies that ACC $\subseteq \mathrm{Dtime}(n^2)^Y$ for some $Y \in$ PP. Since ACC $\subseteq$ PP, suppose for the sake of contradiction that ACC = PP. Then ACC = P = PP. Therefore, $\mathrm{Dtime}(n^3)^Y \subseteq \mathrm{P}^Y \subseteq \mathrm{P} = \mathrm{ACC} \subseteq \mathrm{Dtime}(n^2)^Y$. But this contradicts the time hierarchy theorem of [17]. □

This seems to be one of the very few instances where lower bounds are known for the uniform circuit complexity of certain languages or functions, but where nothing is known about the nonuniform circuit complexity. In fact, the only other instance that we are aware of is that it is not known if EXPTIME contains sets that are not in P/poly (the class of languages accepted by nonuniform circuit families of polynomial size), whereas it does contain sets that are not in P (which is the class of languages accepted by uniform circuit families of polynomial size). In contrast with our results, the combinatorial and algebraic techniques developed in [18], [22], [25] make no use of uniformity, and thus they provide lower bounds on nonuniform circuit size. The uniformity condition is critical in the proof of Theorem 1.1; it is still unknown if PP = Dlogspace-uniform ACC. Although Dlogspace-uniform ACC is trivially seen to be properly contained in PSPACE, it is not known if P-uniform ACC = PSPACE. In fact, it is even unknown if there is any set in Ntime $(2^{n^{O(1)}})$ that is not accepted by a nonuniform ACC circuit family.

To prove Theorem 1.1, we will first use the results of Toda [26], Yao [30], and Beigel and Tarui [10] to convert a circuit family in ACC(subexp) into an equivalent circuit family of depth-two circuits with a symmetric gate at level two, AND gates of small fan-in at level one, and the input gates at level zero. However, since we need the resulting circuit family to be uniform as well, we need to show that the above conversion process can be done uniformly. We then show that the language recognized by the new circuit family can be quickly recognized by a deterministic Turing machine that has access to a particular oracle set in PP. Results about PERM then follow from Valiant's [27] results about the class #P.

Section 2 presents some basic definitions. Section 3 states Theorem 3.1, which is a uniform version of the main result of [10]. Theorem 3.1 is then used to prove the main results of the paper. The final section of the paper presents conclusions and open problems.

The proof of Theorem 3.1, which is the longest and most technically involved part of the paper, is presented in the Appendix. Even though the basic machinery of the proof was developed in [30], [10], there are many obstacles to overcome to ensure that one maintains uniformity.

2. Preliminaries. We will assume that the reader is familiar with circuits and standard complexity classes such as NP, PP, PH, etc., and the various notions of reducibility.

DEFINITION 2.1. *Let m be a positive integer. A* MOD$_m$ gate *outputs* 1 *if the sum of its* (*binary*) *inputs is* 0 *modulo m*; 0, *otherwise. That is,*

$$\mathrm{MOD}_m(x_1,\ldots,x_n)=\begin{cases}1 & \mathit{if} \sum_i x_i \equiv 0 \pmod{m},\\ 0 & \mathit{otherwise.}\end{cases}$$

DEFINITION 2.2. *A* MAJORITY gate *with n inputs outputs* 1 *if $\frac{n}{2}$ or more of its inputs are* 1; 0, *otherwise. That is,*

$$\mathrm{MAJORITY}(x_1,\ldots,x_n)=\begin{cases}1 & \mathit{if} \sum_i x_i \ge \frac{n}{2},\\ 0 & \mathit{otherwise.}\end{cases}$$

DEFINITION 2.3 [21], [4], [7]. *A language L is in* ACC *if there exists a positive integer m such that L is recognized by a family of constant depth polynomial size circuits containing* NOT *gates and unbounded fan-in* AND, OR, *and* MOD$_m$ *gates.*

ACC was first defined and studied in [21], [4], [7] under the name ACC^0. Barrington and Thèrien showed that ACC is equal to the class of languages recognized by polynomial length programs over solvable monoids [7]. Razborov [22] and Smolensky [25] also studied bounded depth circuits containing AND, OR, and MOD gates. Yao's definition of ACC is slightly different from the one given by Barrington et al.; it allows a fixed finite set S of moduli instead of a single modulus m. It is easy to see that a MOD$_m$ gate can simulate a MOD$_k$ gate for any k that divides m. Letting m be the least common multiple of the elements in S makes the two definitions equivalent. Yao [30] showed that every language in ACC is recognized by a family of depth-two probabilistic circuits with a symmetric gate at level two and $2^{(\log n)^{O(1)}}$ AND gates having fan-in $(\log n)^{O(1)}$ at level one. Beigel and Tarui [10] improved this to show the existence of deterministic circuit families of this sort.

DEFINITION 2.4. *For an* NP *machine M, let $\#M$ be the function $\#M : \Sigma^* \to \mathbf{N}$ defined by $\#M(x)$ = number of accepting paths of M on input x. Then $\#P = \{\#M : M$ is an* NP *machine$\}$.*

It is well known from [27] that PERM is complete for #P under polynomial time many-one reductions ($\leq_m^p$). (See also [31], [9].)

DEFINITION 2.5. *A language L is said to be in* PrTime($t(n)$) *if there exists a nondeterministic machine M that runs in time $t(n)$ such that for all $x \in \Sigma^*$, $x \in L \iff$ more than half of the computation paths of M on input x are accepting.*

DEFINITION 2.6. *A language L is said to be in* $\mathrm{C_{=}Time}(t(n))$ *if there exists a nondeterministic machine M that runs in time $t(n)$ such that for all $x \in \Sigma^*$, $x \in L \iff$ exactly half of the computation paths of M on input x are accepting.*

In particular, for polynomial running times we get the well-known classes $\mathrm{PP} = \mathrm{PrTime}(n^{O(1)})$ and $\mathrm{C_{=}P} = \mathrm{C_{=}Time}(n^{O(1)})$.

DEFINITION 2.7. *Let $\{C_n\}$ be a family of circuits. Following* [23], *we define the direct connection language L of $\{C_n\}$ as $L = \{\langle n, g_1, g_2\rangle : g_1 = g_2$ and g_1 is a gate in C_n or $g_1 \neq g_2$ and g_2 is an input to g_1 in $C_n\}$.* Here g_1 and g_2 are names of gates and n is in binary notation.

DEFINITION 2.8. *A circuit family $\{C_n\}$ is* dlogtime-uniform *if its direct connection language can be recognized in* linear *time by a* deterministic *Turing machine. The Turing machine that recognizes the direct connection language of $\{C_n\}$ will be referred to as the* uniformity machine *for $\{C_n\}$.*

The above notion of uniformity is the one that is generally used for small complexity classes (see [6], [12], [23]). However, we are going to use a slightly less restrictive notion of uniformity for our results. Our notion of uniformity can be informally referred to as polylogtime-uniformity. The reason that we use this notion is that we are dealing with circuits of possibly superpolynomial size and the proofs are much simpler with this uniformity condition. It should be noted that a set has uniform ACC(subexp) circuits with respect to our notion of uniformity if and only if it has Dlogtime-uniform ACC(subexp) circuits. This can be established by "padding" a circuit with many dummy gates.

DEFINITION 2.9. *A circuit family $\{C_n\}$ is* uniform *if its direct connection language can be recognized in* polynomial *time by a* deterministic *Turing machine. Note that the time is polynomial with respect to the length of the strings in the language ($|\langle n, g_1, g_2\rangle|$) and not merely polynomial in n.*

DEFINITION 2.10. *Let* ACC($s(n)$) *denote the class of languages accepted by circuit families of constant depth circuits with* NOT *gates and unbounded fan-in* AND, OR, *and* MOD_m *gates (for some integer $m \geq 2$) of size $s(n)$. Then*

$$\mathrm{ACC(subexp)} = \bigcup_{s \in \mathrm{subexp}} \mathrm{ACC}(s(n)),$$

$$\mathrm{ACC(subsubexp)} = \bigcup_{s \in \mathrm{subsubexp}} \mathrm{ACC}(s(n)).$$

Throughout the rest of the paper, classes ACC, ACC(subexp), and ACC(subsubexp) denote *uniform* circuit classes according to the notion of uniformity in Definition 2.9 unless the uniformity condition is mentioned explicitly.

3. The main results. For the proof of Theorem 1.1, we will first show the following.

THEOREM 3.1. *Suppose L is accepted by an ACC(subexp) circuit family. Then L is accepted by a uniform, depth-two circuit family*[1] *of $s(n)$ sized circuits that have the following properties:*

[1] This circuit family is not an ACC(subexp) circuit family because the circuits have arbitrary symmetric gates at their roots. When we say that it is uniform, we are using a slightly different notion of uniformity which is explained in Definition A.22.

1. *Level one consists of a subexponential number of* AND *gates having fan-in* $(\log s(n))^{O(1)}$. *Furthermore, given the name of one of these* AND *gates, the exact fan-in of this AND gate can be computed deterministically in time* $(\log s(n))^{O(1)}$.

2. *There is a symmetric gate at level two. Furthermore, given the number m of* AND *gates that evaluate to one, it can be determined deterministically in time* $(\log s(n))^{O(1)}$ *if the symmetric gate will evaluate to one.*

The above theorem is the most important part of the argument. It is equivalent to saying that the main theorem of [10] holds also in the setting of uniform circuit complexity. Unfortunately, transformations that are obvious in the nonuniform setting require considerable care when undertaken in the uniform setting; we present a complete proof of Theorem 3.1 in the Appendix. The rest of this section assumes that Theorem 3.1 is true and uses it to prove our main results.

Proof of Theorem 1.1. Let $\{C_n\}$ be a circuit family in ACC(subexp) that accepts L. Using the result in Theorem 3.1, we can get a uniform family of circuits $\{D_n\}$ such that for every n, D_n is a deterministic depth-two circuit with the properties mentioned in the statement of Theorem 3.1.

Let M_L be a nondeterministic Turing machine that, on input x, guesses the name of one of the AND gates of D_n ($n = |x|$) and the names of all the inputs of D_n that are connected to this gate. It verifies that the guesses are correct (using the uniformity machine for $\{D_n\}$). It then accepts if and only if the AND gate evaluates to 1 when x is the input to D_n. Since $\{D_n\}$ is uniform and the AND gates have fan-in $o(n^\epsilon)$ (for every ϵ), M_L can do this computation in linear time. Note that $\#M_L(x)$ is the number of AND gates of D_n that evaluate to 1 on input x.

Let $M_1, M_2, \ldots,$ be an enumeration of nondeterministic machines running in linear time. Define the set Y to be $\{\langle i, x, l\rangle : x \in \{0,1\}^* \text{ and } \#M_i(x) > l\}$. Note that Y is in PP[2]. With oracle Y, a deterministic machine (say M) can compute $\#M_L(x)$ in time n^2 using the binary search technique. Then, since this is the number of AND gates of D_n that evaluate to 1 on input x, it can then in linear time determine if D_n accepts x, using the properties guaranteed by Theorem 3.1. Thus membership of x in L can be determined in time n^2 relative to oracle Y. (Note that the running time can actually be brought down to $o(n)$ by modifying the oracle Turing machine model, but we choose not to do so for the sake of clarity.) □

COROLLARY 3.2. *The following statements are true*:

1. ACC(subexp) $\subseteq$ Dtime$(n^9)^{\text{PERM}[1]}$ *where* PERM[1] *refers to the case when only one call is made to* PERM.

2. *There is a set Z in* $C_=P$ *such that* ACC(subexp) $\subseteq$ Ntime$(n^2)^Z$.

Proof. 1. Let M_L and M be the machines from the proof of Theorem 1.1. Note that if M has access to PERM, it can compute $\#M_L(x)$ in time n^9 with just one call to PERM because PERM gives the exact number of accepting paths. The bound n^9 comes from a naïve analysis of Valiant's reduction [27] applied to nondeterministic Turing machines running in linear time.

2. As before, let $M_1, M_2, \ldots,$ be an enumeration of nondeterministic Turing machines running in linear time. Let Z be the set $\{\langle i, x, l\rangle : x \in \{0,1\}^* \text{ and } \#M_i(x) = l\}$. It is not hard to see that Z is in $C_=$P (much like $Y \in$ PP in Theorem 1.1). Let M_L be as above. A nondeterministic machine can compute $\#M_L(x)$ in time n^2 using Z as an oracle. It guesses a value l for $\#M_L(x)$ and asks the appropriate query $\langle i, x, l\rangle$ to Z. □

THEOREM 3.3. ACC $\subsetneq$ $C_=$P.

[2] Let M be a nondeterministic machine that is given input $\langle i, x, l\rangle$. Suppose $t(|x|)$ is the total number of paths of M_i on input x. The computation of M will have $2t(|x|)$ paths; the first $t(|x|)$ of those consist of $t(|x|) - l$ trivially accepting and l trivially rejecting paths, and the other $t(|x|)$ paths will simulate the computation of M_i on x. It is easy to see that $\langle i, x, l\rangle \in Y$ if and only if $\#M_i(x) > l$ if and only if more than half of the paths of M are accepting.

Proof. Corollary 3.2 implies that ACC $\subseteq$ Ntime$(n^2)^Z$ for a set $Z \in$ C$_=$P. Since ACC $\subseteq$ C$_=$P, for the sake of contradiction assume that ACC = C$_=$P. Since co-NP $\subseteq$ C$_=$P and ACC is closed under complement, it follows that ACC = P = NP = C$_=$P. Therefore, Ntime$(n^3)^Z \subseteq$ NP$^Z \subseteq$ NP$^{\text{ACC}}$ = NP$^{\text{P}}$ = NP = ACC $\subseteq$ Ntime$(n^2)^Z$, which contradicts the hierarchy theorem of [15] for nondeterministic time classes. □

THEOREM 3.4. *The permanent function* (PERM) *does not have* ACC(subexp) *circuits.*

Proof. Corollary 3.2 states that ACC(subexp)$\subseteq$ Dtime$(n^9)^{\text{PERM}[1]}$. By the hierarchy theorem of [17], we know that Dtime$(n^9)^{\text{PERM}[1]} \subsetneq$ Dtime$(n^{10})^{\text{PERM}[1]}$. Suppose PERM has ACC(subexp) circuits. Let $L \in$ Dtime$(n^{10})^{\text{PERM}[1]}$ and let M be the oracle machine that accepts L making at most one call to PERM. Let $L' = \{\langle x, z\rangle : M$ accepts x if z is used as the answer to the query made by M to PERM on input x }. Clearly, $L' \in$ P. Similarly, let $L'' = \{\langle x, i\rangle :$ the ith bit of the query by M on input x is 1}. Clearly, $L'' \in$ P as well. A careful reading of Valiant's proof [27] reveals that the membership question for any set in P can be reduced to PERM via uniform AC^0 circuits. (In brief, Valiant's reduction takes an input y to a #P function f, builds a CNF formula ϕ such that $f(y)$ is equal to the number of satisfying assignments to ϕ, and then builds a weighted graph whose permanent is equal to $f(y)$. It has been noted before (e.g., in [19]) that ϕ can be built in uniform AC^0. An inspection of Valiant's graph construction shows that the presence or absence of each edge depends only on the presence of a literal in a given clause, and thus can be computed in uniform AC^0.) Therefore, by the hypothesis, P has ACC(subexp) circuits. Now we can describe an ACC(subexp) circuit family for L. On any input, the query made to PERM is constructed using the circuits for L''; the circuits for PERM are then used to get the answer to the query, and finally, we use the circuits for L' to determine whether $x \in L$. Since L', L'', and PERM all have ACC(subexp) circuit families, the resulting family for L is also in ACC(subexp). Therefore, using the result in Theorem 1.1, $L \in$ Dtime$(n^9)^{\text{PERM}[1]}$, which contradicts the hierarchy theorem of [17] since we started with an arbitrary L in Dtime$(n^{10})^{\text{PERM}[1]}$. □

THEOREM 3.5. PP $\not\subseteq$ ACC(subsubexp).

Proof. We claim that if PP $\subseteq$ ACC(subsubexp), then PrTime(subsubexp) $\subseteq$ ACC(subexp). To see this, note that if $L \in$ PrTime$(t(n))$ for some $t \in$ subsubexp, then $L' \in$ PP, where $L' = \{x10^{t(|x|)} : x \in L\}$. Since by assumption $L' \in$ ACC(subsubexp), one can build subexponential size circuits for L because the composition of two functions in subsubexp is in subexp. This implies that PrTime(subsubexp) $\subseteq$ ACC(subexp).

Note that using the result in Theorem 1.1 and the hierarchy theorem of [17], we know that there are sets in P$^{\text{PP}}$ that are not in ACC(subexp). However, if PP is contained in ACC(subsubexp), then

$$\begin{aligned} \text{P}^{\text{PP}} &\subseteq \text{P}^{\text{ACC(subsubexp)}} \\ &\subseteq \text{P}^{\text{Dtime(subsubexp)}} \\ &\subseteq \text{Dtime(subsubexp)} \\ &\subseteq \text{PrTime(subsubexp)} \\ &\subseteq \text{ACC(subexp)}. \end{aligned}$$

The last step follows from the claim above. Hence, P$^{\text{PP}} \subseteq$ ACC(subexp), which is a contradiction, and the theorem follows. □

Theorem 3.6 is stronger than Theorem 3.5; we include both results to demonstrate the proof technique.

THEOREM 3.6. C$_=$P $\not\subseteq$ ACC(subsubexp).

Proof. We note, as above, that if $\mathrm{C_=P} \subseteq$ ACC(subsubexp), then ACC(subexp) contains $\mathrm{C_=}$Time(subsubexp). We also have that co-$\mathrm{C_=}$Time(subsubexp) $\subseteq$ ACC(subexp) since ACC(subexp) is closed under complement.

Using the result in Corollary 3.2 and the hierarchy theorem of [15] for nondeterministic time, we know that there are sets in $\mathrm{NP^{C_=P}}$ that are not in ACC(subexp). If $\mathrm{C_=P} \subseteq$ ACC(subsubexp), then

$$\begin{aligned}\mathrm{NP^{C_=P}} &\subseteq \mathrm{NP^{ACC(subsubexp)}}\\ &\subseteq \mathrm{NP^{Dtime(subsubexp)}}\\ &\subseteq \mathrm{Ntime(subsubexp)}\\ &\subseteq \mathrm{co\text{-}C_=Time(subsubexp)}\\ &\subseteq \mathrm{ACC(subexp)},\end{aligned}$$

which is a contradiction. □

4. Conclusion. We have shown that uniform ACC circuits of subexponential size cannot compute the permanent function. We have also proved a somewhat weaker bound for some sets in PP. The proofs are based on a simulation of ACC given by Beigel and Tarui in [10]. We have shown how to carry out this simulation in the uniform setting. Some of the obvious open problems are the following.

1. Is uniformity really necessary? Our lower bound proofs work only in the uniform setting. Can we prove a lower bound for the permanent with respect to nonuniform ACC circuits?

2. How powerful are nonuniform ACC circuits? It is still unknown if Ntime $(2^{n^{O(1)}})$ contains sets that are not accepted by nonuniform ACC circuit families.

3. The lower bound that we have for PP is not as strong as the one for permanent. Can it be improved? Even though the permanent function seems to provide more information about the number of accepting paths of NP machines (the permanent gives us all the bits whereas PP only gives us the most significant bit) we still think that a subexponential lower bound can be proved for PP as well.

The work presented here originally started off as the study of sets that are immune to small complexity classes such as $\mathrm{AC^0}$ and ACC. An infinite set L is immune to a complexity class $\mathcal{C}$ if no infinite subset of L is in $\mathcal{C}$. In [1], we show that $\mathrm{P^{PP}}$ contains sets that are immune to ACC, and that nonrelativizing proof techniques suitable for attacking the Dtime vs. Ntime question about exponential time would result from a proof of existence as well as a proof of nonexistence of sets in NP that are immune to $\mathrm{AC^0}$.

It should be emphasized that our results about the complexity of PERM do not rely on any unproven complexity-theoretic assumptions. This is in contrast to other results, such as [14], which proves stronger intractability results about PERM under the hypothesis that the polynomial hierarchy is infinite.

We conclude with a few remarks about some related work that has been done recently. In [16], Green et al. have studied the class of languages that can be recognized in polynomial time with the information about just one bit from the value of a #P function. They define the class MidBitP and show that the classes $\mathrm{MOD}_k\mathrm{P}$, for every k, and the class PH are all low for MidBitP. They have also improved the existing upper bounds for ACC by introducing the idea of MidBit gates. A MidBit gate over w inputs $x_1, x_2, \ldots, x_w$ is a gate that outputs the value of the middle bit in the binary representation of the number $\sum_{i=1}^{w} x_i$. They show that every language in ACC can be accepted by a family of depth-two deterministic circuits of size $2^{(\log n)^c}$ with a MidBit gate at the root and AND gates of fan-in $(\log n)^c$ at the second level.

It would be interesting to see if our techniques can be used in this setting to obtain stronger lower bounds.

Barrington has written an article [5] about the power of circuits of constant depth and $2^{(\log n)^{O(1)}}$ (quasipolynomial) size. The article surveys many results that deal with these kinds of circuits and provides an overview of the new complexity classes that have been introduced. The paper also shows that the notion of uniformity introduced for constant depth circuit families of polynomial size in [6] can be extended to quasipolynomial size as well. It should be noted that this extended notion of uniformity coincides with the one that we have used. Independently of our work, Barrington's paper outlines a proof that shows that the simulation of Beigel and Tarui [10] is uniform according to this new notion of uniformity; thus [5] may be consulted for an alternative approach to proving Theorem 3.1. (The proof in [5] leaves many details to the reader.) In addition, it also shows that the simulation of Green et al. [16] is uniform under this notion as well.

A. Appendix. The appendix is devoted to the proof of Theorem 3.1, which can be regarded as a uniform version of the main theorem of [10]. The definitions, lemmas, and theorems presented in this section all lead up to the proof. Since the proof of Theorem 3.1 is fairly involved, we first start with a very high level outline.

Outline. Since our goal in this section is to prove that the construction of [10] can be done uniformly, it is necessary to prove some preliminary results about uniform constant depth circuits. To that end, we define the notions of "clean" and "nice" circuits, which are circuits that have certain properties that we find essential in presenting our uniformity results. The proof of Theorem 3.1 consists of a number of transformations of a circuit. Without loss of generality, we start out with a "nice" circuit family. After each transformation, we will have a circuit that may not obviously satisfy the "niceness" condition, but at least satisfies the weaker notion of being "clean." We show that this clean circuit can then be transformed into a nice circuit of the same depth, and the process repeats.

The main steps in the transformation are as follows.

1. All the AND and OR gates in the circuits are replaced by constant depth probabilistic subcircuits. This step removes all the OR gates from the circuits and the only remaining AND gates have small fan-in. The circuits are probabilistic but the number of probabilistic bits used in each case is small and is in fact a simple function of the size of C_n.

2. All the MOD gates in the circuit with composite moduli are replaced with equivalent subcircuits so that the resultant circuits consist only of MOD gates with prime moduli.

3. The circuits are now made deterministic by taking separate copies of those for each setting of the probabilistic bits and connecting all outputs to a MAJORITY gate.

4. A general technique is used, showing how nice circuits with small fan-in AND gates can be replaced by equivalent circuits with the same depth, whose outputs are MOD gates.

5. An induction is begun where each step reduces the depth of the circuit. At the beginning of the inductive step, the circuit consists of a symmetric gate on the output level, where the inputs to the symmetric gate are "nice" ACC circuits with MOD$_p$ gates feeding into the symmetric gate. Then, using techniques developed by Toda [26], Yao [30], and Beigel and Tarui [10], we create an equivalent circuit with a new symmetric gate that "absorbs" the level of MOD$_p$ gates; thus the new circuit has smaller depth.

A.1. Nice circuits. In this section, we present a series of "niceness" conditions, and prove that it is no loss of generality to deal only with "nice" circuits.

DEFINITION A.1. *A circuit family $\{C_n\}$ is* well named *if for every n, the name of the output gate of C_n can be computed from n (in binary) in polynomial time (i.e., in $(\log n)^{O(1)}$ time).*

DEFINITION A.2. *A circuit family $\{C_n\}$ is said to have the* strong connection *property if for all n, for every connection $g \rightarrow h$ in C_n, where i is the number such that g is the* ith *input to h*

(assuming lexicographic ordering), it is the case that h can be computed in polynomial time from $\langle n, g, i\rangle$, and additionally, given $\langle n, h, g\rangle$, the number i can be computed in polynomial time. Under the weaker assumption that this condition holds whenever h is an AND *gate then* $\{C_n\}$ *is said to have the* strong connection property for ANDs.

DEFINITION A.3. *A circuit family* $\{C_n\}$ *is said to have* small fan-in AND gates *if for every n, the fan-in of each* AND *gate in C_n is polylogarithmic in the size of C_n.*

DEFINITION A.4. *Let C be a circuit and let P be a path in C from the output gate to an input gate (say t). Let $G_1, G_2, \ldots, G_k$ be the sequence of the types of gates occurring on P so that G_1 is the type of the output gate of C and G_k is the type of the gate that t is connected to. Then the sequence $(G_1, G_2, \ldots, G_k)$ is defined as the* signature *of the path P.*

DEFINITION A.5. *The* compression *of a signature s is the sequence s' that results from applying the following operation as many times as possible to s: replace* "AND, AND" *by* "AND" *and replace* "OR, OR" *by* "OR". *That is, the compression of s contains no two adjacent* ANDs *or* ORs.

DEFINITION A.6. *A circuit family* $\{C_n\}$ *is* clean *if*

1. *It is well named.*
2. *It has the strong connection property for* ANDs.
3. *Every path from an output gate to an input gate in C_n (for every n) has the same signature. (Note that only constant depth circuit families can be clean, since the signature does not depend on n.)*

DEFINITION A.7. *A circuit family* $\{C_n\}$ *of size $s(n)$ is* nice *if it has the following properties.*

1. *It is clean.*
2. *For every n, the fan-in of every gate g in C_n can be computed from g in time* $(\log s(n))^{O(1)}$.
3. *For every n, the depth of a gate g in C_n can be computed from g in time* $(\log s(n))^{O(1)}$.
4. *Each circuit C_n is in tree form (excluding the inputs and negated inputs, which may fan out to many gates at level* 1*).*
5. *It has the strong connection property.*
6. *For all input lengths n, all the* MOD *gates in C_n have the same fan-in.*

Our main lemma in this section is Lemma A.8, which states that any uniform ACC(subexp) circuit family can be transformed into an equivalent nice family.

LEMMA A.8. *Suppose* $\{C_n\}$ *is an ACC*(subexp) *circuit family. Then there exists an equivalent nice family* $\{D_n\}$ *of subexponential size. Furthermore,*

1. *If* $\{C_n\}$ *is clean, then the signature of* $\{D_n\}$ *is the compression of the signature of* $\{C_n\}$.
2. *If* $\{C_n\}$ *is clean and has small fan-in* AND *gates, then* $\{D_n\}$ *has small fan-in* AND *gates.*

The proof of the above lemma follows from a sequence of lemmas that are presented below. The proofs of these lemmas make use of a version of the Alternating Turing Machine (ATM) model of computation. For background on alternation, see [13]. It should be noted that the model that we use here is somewhat different from the one defined in [13]. Some of the lemmas that follow are similar in flavor to the results in [23], in which correspondences between ATMs and uniform circuits were first established; the reader may wish to consult [23].

The following "road map" is intended to explain how the following lemmas combine to prove Lemma A.8.

(i) Lemma A.15: circuit $\mapsto$ ATM.

(ii) Lemma A.16: ATM $\mapsto$ clean ATM.

(iii) Lemma A.17: clean ATM $\mapsto$ nice ATM.
(iv) Lemma A.18: nice ATM $\mapsto$ nice circuits.

The transformations in Lemmas A.15, A.17, and A.18 preserve various properties, such as the property of having small fan-in AND gates.

The existential and universal states in our ATMs behave as usual. Each configuration of an ATM has either zero, one, or two successor configurations (i.e., the fan-out of any node in the computation tree is at most two). We follow the convention that the ATM is always provided the length of the input (in binary) on the work tape as part of its initial configuration on a particular input. This convention has been introduced to simplify the proof.[3] We consider ATMs that access their input only at the leaves. (That is, the only configurations that depend on the input are halting configurations. These are of two types: those that accept if and only if bit i of the input is 1, and those that accept if and only if the complement of bit i is 1 (for some i that is recorded on the address tape). The results in [24] show that this convention can be introduced without loss of generality.)

The MOD states and other aspects of our ATM model are described in the following definitions.

DEFINITION A.9. *For a modulus m, a* MOD$_m$ *configuration* (*say* σ) *is the root of a subtree of associated configurations. This tree is called the* subtree associated with σ *and is represented as T_σ. We say that σ is* accepting *if and only if the number of leaves of T_σ that are accepting is congruent to 0 modulo m. We also use the term* MOD-tree *at times to refer to a subtree associated with a* MOD *configuration.*

DEFINITION A.10. *There is said to be an* alternation *between two configurations σ_1 and σ_2 of an* ATM *if and only if σ_2 follows from σ_1 via one step of the* ATM *and one of the following conditions hold.*

1. *σ_1 is the leaf of a* MOD*-tree, and σ_2 is of type* $\exists$, $\forall$ *or* MOD.
2. *σ_1 is of type $\exists$ and σ_2 is of type $\forall$ or* MOD.
3. *σ_1 is of type $\forall$ and σ_2 is of type $\exists$ or* MOD.

Let T denote the computation tree of an ATM *M on a particular input. The root of the tree is said to have* alternation depth 1. *A node in the tree labeled by configuration σ_2 with parent labeled by configuration σ_1 is defined to have alternation depth one greater than the alternation depth of σ_1 if there is an alternation between σ_1 and σ_2; the alternation depth of σ_2 is equal to that of σ_1 otherwise. The alternation depth of a tree is the maximum alternation depth of all nodes in the tree. The alternation depth of an* ATM *is the maximum alternation depth of all its alternation trees.*

It is necessary for us to define a notion of "clean" ATMs corresponding to our notion of "clean" circuit families. This is accomplished using the following definitions.

DEFINITION A.11. *Let σ and τ be two different configurations of an* ATM. *If τ is reached from σ via a path that contains an alternation only in the step at which τ is reached, then τ is called a* primary descendent *of σ.*

DEFINITION A.12. *For a computation path of an* ATM *on an input, let $C_1, C_2, \dots, C_k$ be the sequence of configurations such that C_1 is the initial configuration, and C_{i+1} is a primary descendent of C_i. The* signature *of the path is the sequence $t_1, t_2, \dots, t_k$ such that if configuration C_i is existential* (*universal,* MOD$_m$), *then $t_i =$* OR (AND, MOD$_m$).

DEFINITION A.13. *An* ATM *is* clean *if every path in every alternation tree of the* ATM *on every input has the same signature.* (*Note that only* ATMs *making $O(1)$ alternations can be clean.*)

[3] It is worthwhile to note that the input length can be computed deterministically in logarithmic time (see [12]) but this requires multiple accesses to the input along a given computation path.

DEFINITION A.14. *An* ATM *running in time $t(n)$ has* well-behaved universal configurations *if each universal configuration has $t(n)^{O(1)}$ primary descendents, and given a universal configuration σ and a number i, the* ith *primary descendent of σ can be computed in time $t(n)^{O(1)}$.*

LEMMA A.15. *Let $L \subseteq \{0, 1\}^*$, let s be a function in subexp, and let L be accepted by a uniform family $\{C_n\}$ of depth d circuits ($d = O(1)$) of type* ACC$(s(n))$. *Then L is accepted by an* ATM *M that has existential* ($\exists$), *universal* ($\forall$), *and* MOD *states (for the same set of moduli) that runs in time* $(\log s(n))^{O(1)}$ *and has alternation depth $a = O(1)$. Moreover,*

1. *If $\{C_n\}$ is clean, then the signature of M is the compression of the signature of $\{C_n\}$.*
2. *If $\{C_n\}$ is clean and has small fan-in* AND *gates, then M has well-behaved universal configurations.*

Proof. Suppose L is accepted by a uniform circuit family $\{C_n\}$. Let U be the uniformity machine for $\{C_n\}$. M behaves as follows.

```
On input $x$, (with $n = |x|$ on the work tape)
    ($\exists$) guess the name of the output gate (say $g$) of $C_n$ of length $(\log s(n))^{O(1)}$.
    Use $U$ to verify that $g$ is indeed a gate in $C_n$ (i.e., check that $U$
    accepts $\langle n, g, g\rangle$).
    ($\forall$) gates $h$ of length $(\log s(n))^{O(1)}$ check that $U$ rejects $\langle n, h, g\rangle$
    (so that $g$ is indeed the output gate).
    Call Eval($g$).

Eval($g$)
    If $g$ is an OR gate then
        ($\exists$) guess $h$ (an input to $g$) of length $(\log s(n))^{O(1)}$.
        If $U$ rejects $\langle n, g, h\rangle$ then reject
        else call Eval($h$).
    If $g$ is an AND gate then
        ($\forall$) guess $h$ (an input to $g$) of length $(\log s(n))^{O(1)}$.
        If $U$ rejects $\langle n, g, h\rangle$ then accept
        else call Eval($h$).
    If $g$ is a MOD$_m$ gate then
        Switch to a MOD$_m$ configuration.
        ($\exists$) guess $h$ (an input to $g$) of length $(\log s(n))^{O(1)}$.
            (This is the subtree associated with the MOD$_m$ configuration.)
        If $U$ rejects $\langle n, g, h\rangle$ then reject
        else call Eval($h$).
    If $g$ is a constant gate then
        Accept iff $g$ is the constant 1 gate.
    If $g$ is an input gate then
        Accept iff the corresponding input is 1.
end (Eval).
```

It is fairly obvious that M accepts x if and only if $C_{|x|}$ evaluates to 1 on input x. Note that M consults its input only at the leaves. It is clear that M makes a constant number of alternations and runs in time $(\log s(n))^{O(1)}$. Indeed, the most time-consuming part of the simulation involves running the uniformity machine U. The constructibility conditions on s are also essential here.

If $\{C_n\}$ is clean, then it is well named, and thus the name of the output gate g can be computed deterministically. Also, since all circuits in $\{C_n\}$ have the same signature, each output gate is of the same type. If the type of the output gate is MOD$_m$, for instance, we can avoid the extra two levels of alternation caused by the processing outside the routine Eval by starting out in a MOD_m configuration, deterministically computing g, existentially guessing h, rejecting if U rejects $\langle n, g, h\rangle$, and otherwise proceeding to Eval(h). The case when the output gate is an AND or OR gate is handled similarly. Thus if $\{C_n\}$ is clean, the signature of M can easily be seen to be the compression of the signature of $\{C_n\}$.

If $\{C_n\}$ has the strong connection property for ANDs and all AND gates have fan-in $(\log s(n))^c$, then instead of universally guessing an input h to an AND gate g, universally guess a number $i \leq (\log s(n))^c$ and deterministically compute the name of the gate h. If M is simulating r consecutive levels of AND gates of C_n, it is not hard to see that each universal configuration of M will have at most $(\log s(n))^{rc}$ primary descendents, and M thus has well-behaved universal configurations.

The other claims of the lemma are easily seen to hold. □

Lemma A.15 does not guarantee the existence of a clean ATM accepting a language when the given circuit family is not already clean. This is remedied by the following lemma.

LEMMA A.16. *If L is accepted by an* ATM *M that makes a constant number of alternations between* $\mathrm{MOD}_{m_1}, \mathrm{MOD}_{m_2}, \ldots, \mathrm{MOD}_{m_k}, \exists$, *and* $\forall$ *states and runs in time $t(n)$ then L is accepted by a* clean ATM *N running in $O(t(n))$ time with a constant number of alternations between* $\mathrm{MOD}_{m_1}, \mathrm{MOD}_{m_2}, \ldots, \mathrm{MOD}_{m_k}, \exists$, *and* $\forall$ *states.*

Proof. Suppose M makes at most λ alternations on any input. Then N has the sequence $\mathrm{MOD}_{m_1}, \mathrm{MOD}_{m_2}, \ldots, \mathrm{MOD}_{m_k}, \exists, \forall$ (repeated λ times) hardwired into its finite control. N simply simulates M but follows the signature in its finite control. If N is trying to simulate a move that does not involve an alternation or that involves moving into a state that has the same type as the next type in its signature, it simply proceeds with the simulation and behaves exactly as M does. In the case of a type mismatch, N behaves as follows.

1. If the next state in the sequence is universal (existential), then it executes a one-ary universal (existential) branch and continues the simulation. (Note that amounts to adding a "dummy" node in the alternating tree.)

2. If the next state in the sequence is a MOD_m state for some m, then it executes a m-way MOD_m branch. It trivially accepts along $m-1$ of these branches (following the signature) and continues the simulation on the remaining one.

It is fairly obvious that N is clean and for every x, N accepts x if and only if M accepts x. □

Our main reason for introducing the ATM model is the following lemma, which enables us to construct "nice" circuits.

LEMMA A.17. *Let $2^{t(n)}$ be constructible, and suppose L is accepted by a clean* ATM *M running in time $t(n)$. Then L is accepted by a clean* ATM *N with the same signature (and hence with the same alternation depth) that runs in time $t(n)^{O(1)}$ and also has the following properties.*

1. *Given a configuration σ on an input of length n, the number of primary descendents of σ is computable from σ in time $t(n)^{O(1)}$.*

2. *Given a configuration σ on an input of length n, the alternation depth of σ is computable from σ in time $t(n)^{O(1)}$.*

3. *Given a configuration σ and number $i \leq$ the number of primary descendents of σ, the* ith *primary descendent of σ (under the usual lexicographic ordering) can be computed in time $t(n)^{O(1)}$ from the encoding of σ.*

4. *All the* MOD *configurations in the computation tree have the same number of primary descendents.*

5. *If M has well-behaved universal configurations, then N also has this property.*

Proof. The proof is very similar to the proof of Lemma A.15. We will need to settle on some convention of encoding paths in an alternation tree, with the property that for every path of length $i \leq t(n)$ in an alternation tree, there is exactly one string of length $2 \cdot t(n)$ that denotes that path. This can easily be accomplished by encoding sequences in {left, right, stop}* in the obvious way; note that there will be many strings that do not correspond to any path in the tree. Similarly, pick some encoding of configurations of M so that any configuration σ of M on inputs of length n has a unique encoding using $c \cdot t(n)$ bits (for some constant c). Again, many strings of length $c \cdot t(n)$ will not correspond to any configuration of M.

N will begin its computation on x by first computing (deterministically) $t(n)$. (Note that this can be done regardless of whether the initial configuration of N is existential, universal, or MOD_m.) If M has well-behaved universal configurations, then let $I(n) = b \log t(n)$ for some constant b; otherwise let $I(n) = t(n)$. (Note that the decision of which value to use for $I(n)$ can be encoded in the finite control of N.) Then N will set σ to be equal to the initial configuration of M, and run the routine Eval(σ).

```
Eval($\sigma$)
    If $\sigma$ is an existential or $\mathrm{MOD}_m$ nonhalting configuration then
        existentially guess strings $w$ of length $2 \cdot t(n)$ and $\tau$ of length $c \cdot t(n)$.
        If $w$ encodes a path from $\sigma$ to configuration $\tau$, where the last step
        in the path involves an alternation (so $\tau$ is a primary descendent of $\sigma$)
            then enter a configuration of the same type as $\tau$ and call Eval($\tau$)
            else call Trivial(reject)
    If $\sigma$ is a universal nonhalting configuration then there are two cases:
        (1) We are simulating a machine $M$ with well-behaved universal
        configurations.
            Universally guess $i \leq b \log t(n)$. Let $\tau$ be the $i$th primary descendent
            of $\sigma$. Call Eval($\tau$).
            (If there is no such $\tau$, then call Trivial(accept).)
        (2) Otherwise.
            Universally guess strings $w$ of length $2 \cdot t(n)$ and $\tau$ of length $c \cdot t(n)$.
            If $w$ encodes a path from $\sigma$ to configuration $\tau$, where the last step in
            the path involves an alternation (so $\tau$ is a primary descendent of $\sigma$)
                then enter a configuration of the same type as $\tau$ and call Eval($\tau$)
                else call Trivial(accept)
    If $\sigma$ is a halting configuration, then
        Accept iff $\sigma$ is accepting. (Note that this may involve accessing the input,
        if $\sigma$ depends on input bit $i$ for some $i$.)
end (Eval).
```

The routine Trivial(d) (for $d \in$ {*accept, reject*}) used in the routine Eval is a simple routine that depends on the number of alternations executed thus far by N in its simulation of M. If the next step in the signature calls for computation of type $\exists$ ($\forall$), then N executes a $2^{(c+2)t(n)}$-way existential (universal) branch, all of which in turn call Trivial(d). If the next step in the signature calls for computation of type MOD_m, and $d =$ *accept* (respectively, $d =$ *reject*), then N enters a MOD_m state, executes a $2^{(c+2)t(n)}$-way existential branch all of which call Trivial(*reject*) (respectively, the first of which calls Trivial(*accept*) and the rest of which call Trivial(*reject*)).

Machine N uses its worktape to record the path in the alternation tree leading to the current configuration. Thus no configuration of N will label two distinct nodes in the alternation tree.

Let us now verify the various properties claimed in the statement of the lemma.

Given σ a configuration of N, one can trace through the path in the alternation tree leading to σ (since this information is recorded in σ). This allows one to compute the alternation depth of σ, as well as to find the configuration τ reached after the last alternation on this path, and compute the number j of moves with fan-out 2 that have occurred along this path between τ and σ. If σ is an $\exists$ or MOD configuration, the number of primary descendents of σ is $2^{(c+2)t(n)-j}$. If σ is a $\forall$ configuration, then this number is $2^{(c+2)I(n)-j}$. In the particular case that σ is a MOD configuration, note that $j = 0$; thus all the MOD configurations have the same number of primary descendents. Furthermore, if σ' is the ith primary descendent of σ, then the number i is encoded in $(c+2)t(n) - j$ consecutive positions in the bit string encoding the path leading to σ', thus enabling us to compute σ' given $\langle n, \sigma, i\rangle$. The other claims of the lemma are easy to verify. □

LEMMA A.18. *Let L be accepted by an* ATM *M satisfying the conditions of Lemma* A.17, *running in time $t(n)$. Then there is a nice* ACC($2^{O(t(n))}$) *circuit family $\{C_n\}$ accepting L, such that the signature of $\{C_n\}$ is the same as the signature of M. Furthermore, if M has well-behaved universal configurations, then $\{C_n\}$ has small fan-in* AND *gates.*

Proof. The proof of this lemma is by a standard simulation of the sort introduced by [23]. The output gate of C_n will be labeled by the initial configuration of N on an input of length n (i.e., with n recorded on the worktape, as per the conventions of our ATM model). The inputs to any gate labeled with configuration σ will be all of the primary descendents of σ. Universal configurations are represented by AND gates, existential configurations by OR gates, and MOD$_m$ configurations by MOD$_m$ gates. Halting configurations are either constant 1 or 0 gates (if they do not depend on the input) or are input gates connected to (negated) input i (if they access input bit i).

It is easily verified that $\{C_n\}$ satisfies the requirements of the lemma. □

Proof of Lemma A.8. This follows immediately from Lemmas A.15, A.16, A.17, and A.18. □

A.2. Transformations on circuits. In this section we prove a general lemma, enabling us to replace gates by equivalent subcircuits. (This, of course, is completely obvious in the nonuniform setting. However, in the uniform setting, where we need the additional property that the fan-in of a circuit be easy to compute, we need all of the "niceness" conditions guaranteed by the preceding section.) Then we apply this lemma to remove OR gates, large fan-in AND gates, and composite MOD gates from ACC circuits.

DEFINITION A.19. *Suppose G is a particular type of gate. Let $\{G_r\}$ denote a family of gates such that the gate G_r is of type G and takes r inputs. Let $\{E_r\}$ be a family of subcircuits so that for every r, E_r takes r inputs and has a single output. We will assume an ordering on the inputs of G_r and E_r and let $x_1, x_2, \ldots, x_r$ denote the inputs to G_r and $y_1, y_2, \ldots, y_r$ denote the inputs to E_r. We say that E_r* replaces *G_r in a circuit C if we remove G_r from C and put E_r in its place in such a way that the output gate of E_r is connected to exactly the gates that G_r is connected to in C, and the inputs to G_r now become inputs to E_r so that for all i, $1 \leq i \leq r$, $x_i = y_i$. In general, when we talk about replacing a gate type G in a circuit, we will mean that all occurrences of G in the circuit are replaced simultaneously.*

LEMMA A.20. *Suppose $\{C_n\}$ and $\{E_r\}$ are nice circuit families. Let G denote a particular type of gate used in the circuits of $\{C_n\}$. For every n, let $\{D_n\}$ denote the circuit family obtained by replacing* all *occurrences of G (of the form G_r for various r) by a subcircuit E_r. Then the circuit family $\{D_n\}$ is clean.*

Proof. It is clear that $\{D_n\}$ is well named and that every path from output to input has the same signature. Thus we need only show that $\{D_n\}$ is uniform and has the strong connection property.

Consider the transformation from C_n to D_n for a particular value of n. Let g (with fan-in r) be an instance of G in C_n and let E_r be the subcircuit that replaces g. Suppose E_r consists of the gates $h_0, h_1, \ldots, h_s$ where h_0 is the output gate of E_r. The names of these gates in the new circuit D_n will be $g\#h_i$, for $0 \leq i \leq s$. Let L_0 be the direct connection language for $\{C_n\}$, L_1 for $\{E_r\}$, and L for $\{D_n\}$. Similarly, let f_0, f_1, and f be the functions that, given $\langle n, g, h\rangle$, compute the number i such that h is the ith input to g in C_n, E_n, and D_n, respectively, and let f_0', f_1', and f' be the related functions that compute h given $\langle n, g, i\rangle$. To accept L, and to compute f, one has to consider the following cases.

1. Strings of the form $\langle n, g, h\rangle$ where neither g nor h are of type G. In this case $\langle n, g, h\rangle \in L \iff \langle n, g, h\rangle \in L_0$. Also, $f(n, g, h) = f_0(n, g, h)$.
2. Strings of the form $\langle n, g\#h, g\#h\rangle$. This is done as follows.
   a. Check that $\langle n, g, g\rangle \in L_0$ and that g has type G.
   b. Compute the fan-in r of g from the description of g.
   c. Check that $\langle r, h, h\rangle \in L_1$.
3. Strings of the form $\langle n, g\#h, g\#h'\rangle$ with $h \neq h'$. This is done as follows.
   a. Check that $\langle n, g, g\rangle \in L_0$ and that g has type G.
   b. Compute the fan-in r of g from the description of g.
   c. Check that $\langle r, h, h'\rangle \in L_1$.
   d. Note that $f(n, g\#h, g\#h') = f_1(n, g\#h, g\#h')$.
4. Strings of the form $\langle n, g', g\#h_0\rangle$ where G is not the type of g'. This is done as follows.
   a. Check that $\langle n, g', g'\rangle$ and $\langle n, g, g\rangle \in L_0$, and that g has type G.
   b. Compute the fan-in r of g from the description of g.
   c. Check that $\langle r, h_0, h_0\rangle \in L_1$ (h_0 is the output gate of E_r).
   d. Check that $\langle n, g', g\rangle \in L_0$.
   e. Note that $f(n, g', g\#h_0) = f_0(n, g', g)$.
5. Strings of the form $\langle n, g\#h, g'\rangle$, where G is not the type of g'. This is done as follows.
   a. Check that $\langle n, g, g\rangle$ and $\langle n, g', g'\rangle$ are in L_0, where g has type G.
   b. Check that $\langle n, g, g'\rangle \in L_0$.
   c. Compute the fan-in r of g from its description.
   d. Check that $\langle r, h, h\rangle \in L_1$.
   e. Compute the number j such that g' is the jth input to g (using the strong connection property).
   f. Let $x_1, x_2, \ldots, x_r$ denote the inputs to E_r. Check that $\langle r, h, x_j\rangle \in L_1$.
   g. Note that $f(n, g\#h, g') = j$.
6. Strings of the form $\langle n, g'\#h, g\#h_0\rangle$ where both g and g' are of type G. This is done as follows.
   a. Check that $\langle n, g, g\rangle$ and $\langle n, g', g'\rangle$ are in L_0.
   b. Compute the fan-in r of g and check that h_0 is the output gate of E_r.
   c. Compute the fan-in r' of g' and check that $\langle r', h, h\rangle \in L_1$.
   d. As in the previous case, check that g is the jth input to g' and that h is connected to input j of $E_{r'}$.

It is not hard to see that all the above cases can be checked within the required time bounds and hence the new circuit family $\{D_n\}$ is uniform as well.

A similar analysis shows that f' can also be computed in time polynomial in the length of its input, and thus $\{D_n\}$ has the strong connection property. □

LEMMA A.21. *Suppose L is accepted by an* ACC(subexp) *family* $\{C_n\}$. *Then L is accepted by a nice probabilistic* ACC(subexp) *circuit family* $\{D_n\}$[4] *such that*

1. $\{D_n\}$ *has no* OR *gates and no* MOD$_m$ *gates for composite modulus m.*
2. $\{D_n\}$ *has small fan-in* AND *gates.*
3. *For every n, the number of probabilistic inputs in D_n is polylogarithmic in the size of D_n.*

[4] Note that the circuits in $\{D_n\}$ are probabilistic and hence also have probabilistic inputs, but when we say D_n we mean the circuit that has n nonprobabilistic inputs. We follow this convention because the proof shows how to convert C_n into D_n for every n.

Proof. By Lemma A.8, we may assume that $\{C_n\}$ is nice.

Let n be fixed. The transformation $C_n \rightarrow D_n$ is carried out by performing the following sequence of steps.

1. By a construction in the proof of Lemma 13 in [2], one can replace the AND and OR gates in the circuit by nice depth-six probabilistic circuits with MOD_2 gates and small fan-in AND gates. (This construction is based on an idea of Valiant and Vazirani in [28]; similar constructions may be found in work by Toda [26] and Kannan, et al. [20].) The size of the new circuit is only polynomially more than that of the old one. If the AND or OR gate being replaced has r inputs, then the probabilistic circuit that replaces it uses $O((\log r)^3)$ random bits. The probabilistic circuits have the property that the probability of error for the whole circuit is less than $\frac{1}{4}$ after all the AND and OR gates have been replaced by these probabilistic circuits, even when the same $O((\log s(n))^3)$ probabilistic bits are fed into the probabilistic inputs of each of these subcircuits. (Even though Allender and Hertrampf discuss space uniformity, it is clear from their proof that the probabilistic circuits are uniform even in our sense of uniformity.) We can now apply Lemma A.20 to prove that the new circuit family (now probabilistic) is clean, and thus by Lemma A.8 there is an equivalent nice circuit family $\{C_n^1\}$. Note that $\{C_n^1\}$ has small fan-in AND gates and has no OR gates.

2. Suppose the circuit C_n^1 contains a MOD_m gate (call it G) where m is composite. Let

$$m = \prod_{i=1}^{t} a_i^{e_i},$$

where $a_i < a_{i+1}$ for all i such that $1 \le i \le t-1$ and for all i, $1 \le i \le t$, a_i is prime and $e_i > 0$. We use the elementary fact that $x \equiv 0 \pmod{m} \Longleftrightarrow x \equiv 0 \pmod{a_i^{e_i}}$ for all i, $1 \le i \le t$ to change G into an AND of $\mathrm{MOD}_{a_i^{e_i}}$'s. Suppose G has r inputs. For each m, the subcircuit family $\{E_r\}$ that replaces the MOD_m gates is easily seen to be nice. The subcircuit E_r has depth two, with an AND gate at the top level and $\mathrm{MOD}_{a_i^{e_i}}$ gates at the bottom level for all i, $1 \le i \le t$. The top level AND gate has fan-in t and is connected to each of the MOD gates at the second level. All the MOD gates at the second level have fan-in r and are all connected to each of the inputs of the gate G. We can now use the result of Lemma A.20 to conclude that the new circuit family is clean. Moreover, the family contains MOD gates with only prime power moduli. The subcircuit E_r, other than its input gates, contains only a constant number of gates that depend on m. Since the original circuit family $\{C_n\}$ only has MOD gates for a fixed set of moduli, the size of the circuit after this step goes up by at most a constant factor. We again use Lemma A.8 to get a nice family of probabilistic circuits $\{C_n^2\}$ that has no composite MOD gates, no OR gates, and small fan-in AND gates.

3. This step eliminates all the MOD gates that have moduli of the form p^e where p is prime and $e > 1$ from C_n^2 and replaces them with subcircuits consisting of AND and MOD_p gates. Suppose C_n^2 contains a MOD_{p^e} gate G for some prime p and $e > 1$. This step uses the following result (for references, see, e.g., [10]): x is congruent to 0 (mod p^e) if and only if each of x, $\binom{x}{p}$, $\binom{x}{p^2}$, $\ldots$, $\binom{x}{p^{e-1}}$ are congruent to 0 (mod p). If $x = \sum_{i=1}^{r} x_i$, then for $1 \le j \le e-1$

$$\binom{x}{p^j} = \binom{x_1+x_2+\cdots+x_r}{p^j} = \sum_{S \subseteq \{1,2,\ldots,r\}, |S|=p^j} \bigwedge_{k \in S} x_k.$$

The subcircuit that replaces G is a three level subcircuit that is described as follows.

(a) The top level consists of an AND gate that has fan-in e.

(b) The middle level consists of e MOD_p gates and each of those is connected to the top level AND gate. For all j, $0 \le j \le e-1$, the jth MOD_p gate outputs 1 if and only if

$\binom{x}{p^j} \equiv 0 \pmod{p}$. If G has fan-in r, then the jth MOD_p gate at this level has fan-in $\binom{r}{p^j}$, one corresponding to each subset of the inputs of size p^j.

(c) The bottom level consists of $\sum_{j=1}^{e-1} \binom{r}{p^j}$ AND gates divided into $e - 1$ groups. For all j, $1 \leq j \leq e - 1$, the jth group consists of $\binom{r}{p^j}$ AND gates, one corresponding to each subset of the inputs of size p^j. The inputs to a particular gate in the jth group are the p^j inputs in the subset to which it corresponds and it fans out to the jth MOD_p gate at the middle level. Note that all the AND gates introduced here have constant fan-in.

It is not hard to see that the subcircuit family described above is nice for every prime power p^e. (The only point that is not completely obvious is checking that the strong connection property holds, but this is straightforward to verify.) Using Lemma A.20 we can now replace every MOD gate with a prime power modulus with a subcircuit that consists only of MOD gates with prime moduli and we now get a clean circuit family that only has AND gates and MOD gates with prime moduli. The size of the subcircuit that replaces a MOD_{p^e} gate is $O(\sum_{j=1}^{e-1} \binom{r}{p^j})$ which is a polynomial in the size of the circuit C_n^2, and thus the new circuit family also has subexponential size. The proof is completed by appeal to Lemma A.8. □

A.3. Circuits with symmetric gates. In order to prove Theorem 3.1 we need to show how to convert an ACC(subexp) circuit family into a uniform deterministic depth two-circuit family that has a symmetric gate at the root and AND gates of small fan-in at the bottom level. So far we have only dealt with ACC type circuits. To proceed, we need to deal with circuits that have arbitrary symmetric gates (but only at the root). However, since most of the results proved so far only deal with uniform ACC type circuits, we need to expand the notion of uniformity a little so that the results can also be used with circuits that have arbitrary symmetric gates at the root. The new notion of uniformity is explained in the following definition.

DEFINITION A.22. *Let $f : \mathbf{N} \to \mathbf{N}$ be a function. Then $\{C_{n,t} : n \in \mathbf{N}, 1 \leq t \leq f(n)\}$ is a* uniform family of ACC sequences *if there is a constant d and a finite set S such that for all n and for all t, $C_{n,t}$ is a circuit of depth d taking inputs from the set $\{x_1, x_2, \ldots, x_n\}$ and having* AND, OR, *and* MOD_m *gates (for $m \in S$) and the direct connection language defined as $\{\langle n, t, g_1, g_2\rangle : g_1 = g_2$ and g_1 is a gate in $C_{n,t}$ or $g_1 \neq g_2$ and g_2 is an input to g_1 in $C_{n,t}\}$ can be recognized in polynomial time. A uniform family of* ACC *sequences $\{C_{n,t} : n \in \mathbf{N}, 1 \leq t \leq f(n)\}$ together with a function* SYM : $\mathbf{N} \times \mathbf{N} \to \{0, 1\}$, *defines a uniform* SYMACC *circuit family $\{D_n\}$ such that for every n,*

1. D_n is a circuit with a symmetric gate at the output level that computes SYM(n, i) *where i is the number of its inputs that evaluate to* 1.

2. The symmetric gate has fan-in $f(n)$ and the output gates of $C_{n,t}$, $1 \leq t \leq f(n)$, are connected to it.

3. Given n and i, $f(n)$ and SYM(n, i) *can be computed in time polylogarithmic in the size of D_n.*

Note that the results proved so far also hold with this new notion of uniformity. In particular, letting $f(n) = 1$ for all n and letting SYM be the identity function reduces this to the old notion of uniformity. Also, we will use the fact that Lemma A.8 also holds in this new setting. That is, given a uniform *clean* family of ACC sequences, there is an equivalent *nice* family of ACC sequences with the same signature and of approximately the same size. (In proving the analog of Lemma A.8 in this new setting, the index t of circuit $C_{n,t}$ would be provided to the ATM as an additional parameter on the worktape, along with n.)

LEMMA A.23. *Let L be accepted by an* ACC(subexp) *circuit family $\{C_n\}$. Then there is a constructible subexponential function s and there is a constant c such that L is accepted by a deterministic circuit family $\{D_n\}$ where for every n, D_n has a* MAJORITY *gate at the root, connected to the output gates of $C_{n,t}$, $1 \leq t \leq 2^{(\log s(n))^c}$ where $\{C_{n,t} : n \in \mathbf{N},$

$1 \le t \le 2^{(\log s(n))^c}\}$ *is a uniform family of* ACC *sequences with small fan-in* AND *gates, no* OR *gates, and no* MOD_m *gates for composite* m.

Proof. By Lemma A.21, if L is accepted by an ACC(subexp) circuit family, then L is accepted by a nice ACC(subexp) family of *probabilistic* circuits with small fan-in AND gates, no OR gates, and no MOD_m gates for composite m, using at most $(\log s(n))^c$ probabilistic bits (for some constant c), where $s(n)$ bounds the size of C_n.

Now construct the sequence of circuits $\{C_{n,t}\}$ where t is a bit string of length $(\log s(n))^c$. The gates in $\{C_{n,t}\}$ will have names of the form $\langle t, g\rangle$ where g is a gate in C_n, and the connections among all gates are the same, except that if gate g in C_n is connected to probabilistic bit number j, then gate $\langle t, g\rangle$ will be connected to the jth bit of t. (i.e., the new circuit sequence consists of identical copies of C_n, with particular choices of probabilistic bits hardwired in.)

Let D_n consist of a MAJORITY gate with inputs from the various $C_{n,t}$. It is clear that the new circuit accepts the same language as $\{C_n\}$. The size of D_n is $O(s(n)2^{(\log s(n))^c})$, which is subexponential. It is immediate that the other required properties also hold. □

The following lemma shows how one can in effect "push" an AND gate below a level of MOD gates (much as multiplication distributes over addition).

LEMMA A.24. *Let* $\{C_{n,t}\}$ *be a nice family of* ACC *sequences of subexponential size, having small fan-in* AND *gates, no* OR *gates, and no* MOD_m *gates where m is composite, where the output gate of each circuit is an* AND *gate, and the inputs to that* AND *gate are* MOD_p *gates. Then there is an equivalent nice sequence* $\{D_{n,t}\}$ *with the same depth, also of subexponential size with small fan-in* AND *gates, no* OR *gates, and no* MOD_m *gates where m is composite, where the output gate of each circuit is a* MOD_p *gate, and the inputs to that* MOD_p *gate are* AND *gates.*

Proof. Our proof again follows the outline given in [10] (see also [2], [20]), where we must be careful to see that the transformation can be done uniformly.

Suppose G is an AND gate (the output gate of some $C_{n,t}$ that has r MOD_p gates $G_1, G_2, \dots, G_r$ as inputs). Note that r is polylogarithmic in $s(n)$, where $s(n)$ bounds the size of $C_{n,t}$. Since the sequence $C_{n,t}$ is nice, all the MOD_p gates $G_1, G_2, \dots, G_r$ have the same fan-in. Let this fan-in be denoted by n_0 and let $\{x_{ij}\}$, $1 \le j \le n_0$ denote the set of inputs to G_i. Finally, let $x_i = \sum_{1\le j\le n_0} x_{ij}$. Consider the AND of $G_1, G_2, \dots, G_r$. By Fermat's Little Theorem, for $1 \le i \le r$,

$$1 - x_i{}^{p-1} \equiv \begin{cases} 0 \pmod p & \text{if } x_i \not\equiv 0 \pmod p, \\ 1 \pmod p & \text{otherwise.} \end{cases}$$

Therefore,

$$\bigwedge_{i=1}^{r}[x_i \equiv 0 \pmod p] \Longleftrightarrow 1 - \prod_{i=1}^{r}(1 - x_i{}^{p-1}) \equiv 0 \pmod p.$$

Note that $1 - \prod_{i=1}^{r}(1 - x_i{}^{p-1})$ is a polynomial of degree $r(p-1)$ in the variables x_{ij}, $1 \le i \le r$, $1 \le j \le n_0$. Let $[r]$ denote the set $\{1, 2, \dots, r\}$.

$$\begin{aligned} 1 - \prod_{i=1}^{r}(1 - x_i{}^{p-1}) &= 1 - \left(\prod_{i=1}^{r}\left(1 - \left(\sum_{j=1}^{n_0} x_{ij}\right)^{p-1}\right)\right) \\ &= \sum_{k=1}^{r} \sum_{I\subseteq[r],|I|=k} (-1)^{k-1} \prod_{i\in I}\left(\sum_{j=1}^{n_0} x_{ij}\right)^{p-1} \end{aligned}$$

$$= \sum_{k=1}^{r}(-1)^{k-1} \sum_{I\subseteq[r],|I|=k} \prod_{i\in I} \left(\sum_{j=1}^{n_0} x_{ij}\right)^{p-1}$$

$$= \sum_{k=1}^{r}(-1)^{k-1} \sum_{I\subseteq[r],|I|=k} \prod_{i\in I} \sum_{J_i=\langle j_{i,1},j_{i,2},\ldots,j_{i,p-1}\rangle\in[n_0]^{p-1}} \prod_{l=1}^{p-1} x_{i j_{i,l}}$$

$$= \sum_{k=1}^{r}(-1)^{k-1} \sum_{I\subseteq[r],I=\{i_1,i_2,\ldots,i_k\}} \sum_{J_{i_1},J_{i_2},\ldots,J_{i_{k}k}} \prod_{s=1}^{k}\prod_{l=1}^{p-1} x_{i_s j_{i_s,l}} \quad (*).$$

This expression can be realized by a MOD_p gate (call it g) with AND gates of fan-in at most $r(p-1)$ as inputs. Since r is $(\log s(n))^{O(1)}$, the fan-in of these AND gates is also polylogarithmic in $s(n)$. The only things we need to take care of are the negative coefficients in the above expression. That is done by multiplying[5] the negative coefficients by $(1-p)$. The expression is changed slightly and the term $(-1)^{k-1}$ is replaced by c_k where $c_k = 1$ if k is even and $c_k = p-1$ if k is odd. Now we interpret scalar multiplication as repeated addition and the multiplication of variables is realized by AND gates. From the expression $(*)$, it is not hard to see that the number of AND gates that are input to the new MOD_p gate g is $\sum_{k=1}^{r} c_k\binom{r}{k}(n_0^{p-1})^k$. Note that since r is polylogarithmic in $s(n)$, this expression (i.e., the fan-in of the new MOD gate) can be computed in time $(\log s(n))^{O(1)}$ from $\langle G, G_1, \ldots, G_r\rangle$.

To show that this step can be done uniformly, we must show how the new gates created in this step should be named so that the direct connection language of the new circuit family can be recognized within the required time bound. The name of the new MOD_p gate is $g = \langle G\#\langle G_1, G_2, \ldots, G_r\rangle, \mathrm{MOD}_p\rangle$. Looking at the expression $(*)$, it is clear that a typical AND gate has $k(p-1)$ inputs where $1 \le k \le r$. The $k(p-1)$ inputs can be divided up into k groups of size $(p-1)$ each. Every group represents a distinct gate in the set $\{G_1, G_2, \ldots, G_r\}$. The $(p-1)$ inputs in a particular group (representing say G_i) are simply some of the input gates to G_i (with repetitions allowed) in $C_{n,t}$. Depending on the value of c_k, such an AND gate either appears once or $(p-1)$ times. The name of such an AND gate is $\langle G\#\langle\langle H_1\#L_1\rangle, \langle H_2\#L_2\rangle, \ldots, \langle H_k\#L_k\rangle\rangle\#m, \mathrm{AND}\rangle$ where $H_1, H_2, \ldots, H_k$ are distinct gates from the set $\{G_1, G_2, \ldots, G_r\}$ and each L_i, $1 \le i \le k$ is a list of $(p-1)$ of the gates that are input to H_i in the original circuit. Note that L_i is allowed to have repetitions. The number m is either 0 (indicating only one copy of the gate; this will be the case if k is even) or between 1 and $(p-1)$ and is used for indexing the $(p-1)$ different copies.

We now show how to recognize the direct connection language for the new circuit family that we get after applying this transformation. Let L_0 be the direct connection language before the step and L the one after the step. Note that the strings in L conform to the naming scheme discussed above. The following cases must be considered.

1. Let g be a new[6] gate. To check if $\langle n, t, g, g\rangle \in L$, we have the following two subcases.
   (a) g is a new MOD_p gate of the form $\langle G\#\langle G_1, G_2, \ldots, G_r\rangle, \mathrm{MOD}_p\rangle$. We do the following:
      i. Check that G is the output gate of $C_{n,t}$. (This can be done because the circuits are well-named.)
      ii. Check that $\langle n, t, G, G_i\rangle \in L_0$ for all i, $1 \le i \le r$, where r is the fan-in of G. (Recall that r can be computed from G, by one of the niceness properties.)

[5] Note that this does not change the value of the expression mod p.

[6] The word "new" will hereafter be used to refer to gates that were created in the current step.

(b) g is a new AND gate of the form

$$\langle G\#\langle\langle H_1\#L_1\rangle, \langle H_2\#L_2\rangle, \ldots, \langle H_k\#L_k\rangle\rangle\#m, \text{AND}\rangle.$$

We do the following:

i. Check that G is the output gate of $C_{n,t}$.
ii. Verify that $H_1, H_2, \ldots, H_k$ are all distinct.
iii. Check that for all i, $1 \leq i \leq k$, $\langle n, t, G, H_i\rangle \in L_0$.
iv. Check that m has the right value based on the parity of k.
v. For all i, $1 \leq i \leq k$, verify that L_i is indeed a list of $(p-1)$ gates that are all input to H_i.

2. Let g_1 be an old gate and g_2 a new gate. Then $\langle n, t, g_1, g_2\rangle \notin L$.

3. Let g_1 be a new gate and g_2 an old gate. The only way for $\langle n, t, g_1, g_2\rangle \in L$ to hold is that g_1 is a new AND gate created in this step. Hence g_1 has the form $\langle G\#\langle\langle H_1\#L_1\rangle, \langle H_2\#L_2\rangle, \ldots, \langle H_k\#L_k\rangle\rangle\#m$, AND$\rangle$. We do the following to check if $\langle n, t, g_1, g_2\rangle \in L$.

a. Check that $\langle n, t, g_1, g_1\rangle \in L$.
b. Check that $\langle n, t, g_2, g_2\rangle \in L_0$.
c. Verify that $\exists\, i$, $1 \leq i \leq k$, such that g_2 belongs to the list of gates L_i.

4. Let g_1 and g_2 both be new gates. The only way for g_2 to be an input to g_1 is if $g_1 = \langle G\#\langle G_1, G_2, \ldots, G_r\rangle, \text{MOD}_p\rangle$ and $g_2 = \langle G\#\langle\langle H_1\#L_1\rangle, \langle H_2\#L_2\rangle, \ldots, \langle H_k\#L_k\rangle\rangle\#m, \text{AND}\rangle$ where $\{H_1, \ldots, H_k\} \subseteq \{G_1, \ldots, G_r\}$. This is obviously easy to check.

The only remaining property that needs to be checked is the strong connection property for ANDs. However, this is immediate using the naming system that we use, since the name of each new AND gate explicitly lists the names of each of its inputs.

Let us now consider the size of the new circuit after a single level of AND gates has been pushed below a level of MOD gates. The increase in size comes mainly because of all the new AND gates that get created. For a circuit of size s, the number of new AND gates created to change an AND of r MOD$_p$ gates is $\leq \sum_{k=1}^{r} c_k\binom{r}{k}s^{(p-1)k} \leq (p-1)2^r s^{(p-1)r}$. Therefore, the overall size of the new circuit is at most $O(2^r s^{(p-1)r+1})$. Since s is subexponential and r is polylogarithmic in s, the size of the new circuits is still subexponential.

Note that this step does not preserve the tree structure of the circuit so we use Lemma A.8 to produce an equivalent nice circuit sequence. □

LEMMA A.25. *Let L be accepted by a uniform nice* SYMACC *circuit family $\{C_n\}$ of subexponential size, with small fan-in* AND *gates, no* OR *gates, and no* MOD$_m$ *gates for composite m, such that each path from the output gate to an input passes through $k \geq 1$* MOD *gates. Then there is an equivalent* SYMACC *circuit family $\{D_n\}$ satisfying the same conditions, such that each path from the output gate to an input gate passes through $k-1$* MOD *gates.*

Proof. Our proof follows the outline in [10], using techniques developed in [30], [26].

Let L and $\{C_n\}$ be as in the statement of the lemma, where the output gate of C_n computes the function $A(n, \eta)$, where η is the number of elements of $\{C_{n,i} : 1 \leq i \leq f(n)\}$ that evaluate to 1. By Lemma A.24, we may assume without loss of generality that the output of each circuit $C_{n,i}$ is a MOD$_p$ gate. Since $\{C_n\}$ is nice, for each n there is some n_0 so that each MOD$_p$ gate in $C_{n,t}$ has fan-in n_0 (where n_0 can be computed in $(\log s(n))^{O(1)}$ time from n). For each $i \leq f(n)$, let the inputs to the ith of these MOD$_p$ gates be denoted by $x_{i,j}$, $1 \leq j \leq n_0$. Then the value of $\{C_n\}$ can be expressed as $A(n, \sum_{1\leq i\leq f(n)} \text{MOD}_p(x_{i,1}, x_{i,2}, \ldots, x_{i,n_0}))$.

Let $k(n) = 1 + \lfloor \log_p f(n) \rfloor$ so that $p^{k(n)} > f(n)$. Note that $k(n)$ is computable in time $(\log s(n))^{O(1)}$. For the rest of this discussion, fix n, and let k denote $k(n)$.

It is shown in [10] that the polynomial P_k defined by

$$P_k(y) = (-1)^{k+1}(y-1)^k \left(\sum_{j=0}^{k-1} \binom{k+j-1}{j} y^j \right) + 1$$

satisfies the property that for every $m \geq 1$ and $y \geq 0$,

$$y \equiv 0 \pmod{m} \Longrightarrow P_k(y) \equiv 0 \pmod{m^k}$$

and

$$y \equiv 1 \pmod{m} \Longrightarrow P_k(y) \equiv 1 \pmod{m^k}.$$

Let $Q_k(y) = 1 - P_k(y^{p-1})$. Then

$$Q_k(y) \equiv \begin{cases} 1 \pmod{p^k} & \text{if } y \equiv 0 \pmod{p}, \\ 0 \pmod{p^k} & \text{otherwise.} \end{cases}$$

If $y = \sum_{i=1}^r y_i$ then

$$Q_k\left(\sum_{i=1}^r y_i\right) \equiv \mathrm{MOD}_p(y_1, y_2, \ldots, y_r) \pmod{p^k}.$$

Thus, recalling that the value of the circuit C_n is

$$A\left(n, \sum_{1 \leq i \leq f(n)} \mathrm{MOD}_p(x_{i,1}, x_{i,2}, \ldots, x_{i,n_0})\right),$$

we see that this can also be expressed as

$$A\left(n, \sum_{1 \leq i \leq f(n)} \left(Q_k\left(\sum_{1 \leq j \leq n_0} x_{i,j}\right) \pmod{p^k}\right)\right).$$

Since $f(n) < p^k$ and Q_k is always 0 or 1 $\pmod{p^k}$, we can bring the outer sum inside the modulus to obtain the equivalent expression

$$A\left(n, \left(\sum_{1 \leq i \leq f(n)} Q_k\left(\sum_{1 \leq j \leq n_0} x_{i,j}\right)\right) \pmod{p^k}\right).$$

Let $B(n, i)$ be defined to be $A(n, (i \bmod p^k))$. Thus the value of $\{C_n\}$ is equal to

$$B\left(n, \sum_{1 \leq i \leq f(n)} Q_k\left(\sum_{1 \leq j \leq n_0} x_{i,j}\right)\right).$$

Note that B is computable in time polylogarithmic in $s(n)$.

Note that $(\sum_{1 \leq i \leq f(n)} Q_k(\sum_{1 \leq j \leq n_0} x_{i,j}))$ is a low degree polynomial in the variables $\{x_{i,j}\}$. As in the proof of Lemma A.24, our strategy will be to implement scalar multiplication with AND gates, and multiply the negative coefficients by $(1 - p^k)$ to make them positive[7], to obtain a realization of this polynomial in terms of circuits.

[7] This does not change the value of the expression mod p^k.

Our first task is to compute the coefficients in the polynomial

$$\left(\sum_{1\le i\le f(n)} Q_k\left(\sum_{1\le j\le n_0} x_{i,j}\right)\right).$$

Since this is just a sum of $f(n)$ similar polynomials, we can consider each of them separately.

Recall that $Q_k(y) = 1 - P_k(y^{p-1})$. Let $z = y^{p-1}$. After a little simplification we get $1 - P_k(z) = (1-z)^k(\sum_{j=0}^{k-1}\binom{k+j-1}{j}z^j)$. This is a polynomial of degree $2k-1$. For $i \ge 0$, let $b_i = 1$ if i is even, and $b_i = p^k - 1$ if i is odd. The coefficients of z^m, say c_m, are given by

$$c_m = \begin{cases} 1 & \text{if } m = 0, \\ 0 & \text{if } 1 \le m \le k-1, \\ \displaystyle\sum_{0\le i\le k, 0\le j\le k-1, i+j=m} b_i\binom{k}{i}\binom{k+j-1}{j} & \text{if } k \le m \le 2k-1. \end{cases}$$

These coefficients c_m can be computed in $(\log s(n))^{O(1)}$ time, because we only need to compute $O(k)$ binomial coefficients, each involving numbers that are $O(k\log k)$ bits long. It can be verified that $O(k^4\log k)$ time suffices, which is polylogarithmic in the size of the circuit since k is logarithmic in the size.

Now observe that the value of circuit $\{C_n\}$ is given by

$$\begin{aligned} B\left(n, \sum_{i=1}^{f(n)} Q_k\left(\sum_{j=1}^{n_0} x_{i,j}\right)\right) &= B\left(n, \sum_{i=1}^{f(n)} 1 - P_k\left(\left(\sum_{j=1}^{n_0} x_{i,j}\right)^{p-1}\right)\right) \\ &= B\left(n, \sum_{i=1}^{f(n)}\sum_{m=0}^{2k-1} c_m\left(\sum_{j=1}^{n_0} x_{i,j}\right)^{(p-1)m}\right) \\ &= B\left(n, \sum_{i=1}^{f(n)}\sum_{m=0}^{2k-1}\sum_{c=1}^{c_m}\sum_{\langle j_1, j_2, \ldots, j_{p-1}\rangle\in[n_0]^{(p-1)m}} \bigwedge_{l=1}^{(p-1)m} x_{i,j_l}\right). \end{aligned}$$

In place of each circuit $C_{n,t}$ in the original sequence of circuits, there will be several new circuits, each of the form $D_{n,\langle i,m,c,j_1,\ldots,j_{(p-1)m}\rangle}$ where $0 \le m \le 2k-1$, $1 \le c \le c_m$, and each j_l is in $[n_0]$. (Of course, by our conventions, there will also be circuits $D_{n,t}$ where t is not of this form; each such circuit $D_{n,t}$ will be a trivial rejecting circuit that will therefore have no effect on the output of the symmetric gate.)

The output gate of each circuit $D_{n,\langle i,m,c,j_1,\ldots,j_{(p-1)m}\rangle}$ will be an AND gate with the name $g = \langle n, i, m, c, j_1, \ldots, j_{(p-1)m}, \text{AND}\rangle$. The inputs to g will be the $(p-1)m$ gates that are the j_lth inputs to the MOD$_p$ gate G_i in the original circuit $C_{n,i}$. Note that since $C_{n,i}$ has the strong connection property, one can show that $D_{n,\langle i,m,c,j_1,\ldots,j_{(p-1)m}\rangle}$ does, too.

Note that the number of bits needed to express $\langle i, m, c, j_1, \ldots, j_{(p-1)m}\rangle$ is bounded by $(\log s(n))^b$ for some constant b, and thus if we define $f'(n)$ to be equal to $2^{(\log s(n))^b}$, it follows that the symmetric gate computing B in circuit D_n has fan-in $f'(n)$, where $f'(n)$ is computable in time polylogarithmic in $s(n)$.

Since the new circuit consists of a subexponential number of circuits, each of which is of subexponential size, the new circuit is also of subexponential size.

The depth of the new circuit family is the same as $\{C_n\}$ but the top layer of MOD$_p$ gates has been "absorbed" into the symmetric gate computing B and been replaced by a layer of

AND gates of small fan-in. Now, by an appeal to Lemma A.8, the circuit can be converted into nice form, which completes the proof. □

Proof of Theorem 3.1. By Lemma A.23, every language in ACC(subexp) is accepted by a deterministic SYMACC circuit family of subexponential size, with small fan-in AND gates, no OR gates, and no MOD_m gates for composite m. Successive applications of Lemma A.24 and Lemma A.25 remove all MOD gates from the circuit, while maintaining the property that all AND gates have small fan-in. This suffices to prove the theorem. □

Acknowledgments. We thank the anonymous referees for suggestions that improved the presentation of these results. We also thank Richard Beigel for suggesting that our results concerning PP could be extended to $C_=P$.

REFERENCES

[1] E. Allender and V. Gore, *On strong separations from* AC^0, in Advances in Computation Theory, Jin-Yi Cai, ed., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 13, American Mathematical Society, Providence, RI, pp. 21–37, 1993.

[2] E. Allender and U. Hertrampf, *Depth reduction for circuits of unbounded fan-in*, Inform. Comput., to appear.

[3] J. Aspnes, R. Beigel, M. Furst, and S. Rudich, *The expressive power of voting polynomials*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1991, pp. 402–409.

[4] D. Barrington, *Bounded-width polynomial-size branching programs recognize exactly those languages in* NC^1, J. Comput. System Sci., 38 (1989), pp. 150–164.

[5] ———, *Quasipolynomial size circuit classes*, in Proc. 7th Annual IEEE Structure in Complexity Theory Conference, IEEE Computer Society Press, Washington, DC, 1992, pp. 86–93.

[6] D. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, J. Comput. System Sci., 41 (1990), pp. 274–306.

[7] D. Barrington and D. Thérien, *Finite monoids and the fine structure of* NC^1, J. Assoc. Comput. Mach., 35 (1988), pp. 941–952.

[8] P. Beame, S. Cook, and H. Hoover, Log *depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[9] A. Ben-Dor and S. Halevi, *Zero-one permanent is #P-complete, a simpler proof*, in Proc. 2nd Israel Symposium on Theory of Computing and Systems, IEEE Computer Society Press, Washington, DC, 1993.

[10] R. Beigel and J. Tarui, *On* ACC, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, DC, 1991, pp. 783–792.

[11] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–743.

[12] S. Buss, S. Cook, A. Gupta, and V. Ramachandran, *An optimal parallel algorithm for formula evaluation*, SIAM J. Comput., 21 (1992), pp. 755–780.

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

[14] U. Feige and C. Lund, *On the hardness of computing the permanent of random matrices*, in Proc. 24th Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1992, pp. 643–654.

[15] M. Fischer, A. Meyer, and J. Seiferas, *Separating nondeterministic time complexity classes*, J. Assoc. Comput. Mach., 25 (1978), pp. 146–167.

[16] F. Green, J. Köbler, K. Regan, T. Schwentick, and J. Torán, *The power of the middle bit of a #P function*, J. Comput. System Sci., to appear. Preliminary versions appeared in Proc. 7th Annual IEEE Structure in Complexity Theory Conference, IEEE Computer Society Press, Washington, DC, 1992, pp. 111–117, and in Proc. 4th Italian Conference on Theoretical Computer Science, World Scientific Press, Singapore, 1992, pp. 317–329.

[17] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[18] J. Håstad, *Computational Limitations for Small Depth Circuits*, MIT Press, Cambridge, MA, 1987.

[19] N. Immerman, *Languages that capture complexity classes*, SIAM J. Comput., 16 (1987), pp. 760–778.

[20] R. Kannan, H. Venkateswaran, V. Vinay, and A. Yao, *A circuit-based proof of Toda's theorem*, Inform. Comput., 104 (1993), pp. 271–276.

[21] P. McKenzie and D. Thérien, *Automata theory meets circuit complexity*, in Proc. 16th Annual International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, Vol. 372, Springer-Verlag, Berlin, New York, 1989, pp. 589–602.

[22] A. Razborov, *Lower bounds for the size of circuits of bounded depth with basis* $\{\wedge, \oplus\}$, Math. Notes, 41 (1987), pp. 333–338.

[23] W. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 21 (1981), pp. 365–383.

[24] M. Sipser, *Borel sets and circuit complexity*, in Proc. 15th Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1983, pp. 61–69.

[25] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1987, pp. 77–82.

[26] S. Toda, PP *is as hard as the polynomial-time hierarchy*, SIAM J. Comput., 20 (1991), pp. 865–877.

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

[28] L. Valiant and V. Vazirani, NP *is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.

[29] A. Yao, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, DC, 1985, pp. 1–10.

[30] ———, *On* ACC *and threshold circuits*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, DC, 1990, pp. 619–627.

[31] V. Zankó, *#P-completeness via many-one reductions*, Internat. J. Found. Comput. Sci., 2 (1991), pp. 77–82.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 1050–1074, October 1994

011

THE JOINT DISTRIBUTION OF ELASTIC BUCKETS IN MULTIWAY SEARCH TREES*

WILLIAM LEW† AND HOSAM M. MAHMOUD‡

Abstract. Random search trees are studied when they grow under a general computer memory management scheme. In a general scheme, the space is released in buckets of certain predesignated sizes. For a search tree with branch factor m, the nodes may hold up to $m-1$ keys. Suppose the buckets of the memory management scheme that can hold less than m keys have key capacities $c_1, \dots, c_p$. The search tree must then be implemented with multitype nodes of these capacities. After n insertions, let $X_n^{(i)}$ be the number of buckets of type i (i.e., of capacity c_i, $1 \le i \le p$). The multivariate structure of the tree is investigated. For the vector $\mathbf{X}_n = (X_n^{(1)}, \dots, X_n^{(p)})^T$, the asymptotic mean and covariance matrix are determined. Under practical memory management schemes, all variances and covariances experience a phase transition: For $3 \le m \le 26$, all variances and covariances are asymptotically linear in n; for higher branch factors the variances and covariances become a superlinear (but subquadratic) function of n. The joint distribution of $\mathbf{X}_n$ is shown to be multivariate normal in a range of m. While the tree is growing, conversions between types are necessary. A multivariate problem concerning these conversions with an asymptotic multivariate normal distribution is also studied. The fixed bucket, exact fit, and buddy system allocation schemes will serve as illustrating examples.

Key words. searching, random trees, multivariate statistics

AMS subject classifications. 05C05, 60F05, 68P05

1. Introduction.

The *m-ary search tree* is a data structure that grows by the progressive insertion of keys into a tree with branch factor m. The subtrees are numbered 1 to m, from left to right. The tree is constructed by placing up to $m-1$ keys in its root, sorted from left to right, and then guiding a subsequent key to the jth subtree, $1 \le j \le m$, if that key is greater than exactly $j-1$ of the root keys. In the jth subtree, the newcomer is subjected recursively to the same procedure until a unique insertion position is found.

The usual data probability model is the *random permutation model* where all permutations of $\{1, \dots, n\}$ are considered equally likely input sequences (see [12]). The random permutation model covers a wide variety of situations, since clearly the only data aspect relevant to the structure of the tree is the order statistics; the ranks of data taken from any continuous distribution follows the random permutation model (see [17] for a detailed discussion). For the rest of this paper the term *random tree* will refer to a tree constructed from a random permutation.

In practice, a random tree grows under a computer memory management scheme. Memory management schemes allocate space in blocks of storage called buckets or pages. Sometimes the buckets come in fixed allowed sizes. For example, the buddy system releases blocks of sizes 2^k words of memory, $k = 1, 2, \dots$. We shall assume a general scheme under which the buckets are measured in terms of key capacity. It is clear from the insertion algorithm that the largest buckets that may be used as a tree node must be capable of holding $m-1$ keys and m pointers. Thus the design parameter m must be chosen in an application so that the space for $m-1$ keys, m pointers, and any space needed for system use coincide with the capacity of a system's bucket. It should also be clear that a node with less than $m-1$ keys has no descendants and thus may be implemented without pointers as it is a terminal node. Also some terminal nodes may be holding $m-1$ keys, but they do not need pointers as well. Nonterminal nodes must have pointers.

*Received by the editors April 27, 1992; accepted for publication (in revised form) May 28, 1993.

†The IBM Corporation, 6710 Rockledge Drive, Bethesda, Maryland 20827.

‡Department of Statistics/Statistical Computing, The George Washington University, Washington, D.C. 20052 (hosam@gwuvm.bitnet).

This author's research was supported in part by National Security Agency grant MDA904-92-H-3086.

Let $c_1, c_2, \ldots, c_p$ be all the bucket sizes (in terms of keys) that are less than or equal to $m-1$, with

$$c_1 < c_2 < \cdots < c_{p-1} \le c_p = m - 1. \tag{1.1}$$

We shall call a bucket (a tree node) of capacity c_i, a bucket of type i, $1 \le i \le p$. It is also helpful to think of **NIL** pointers as buckets of type 0 with capacity $c_0 = 0$. The tree thus grows by allocating a bucket of type 1 when insertion falls in an empty subtree. Subsequent insertions that fall in the same bucket will increase the number of keys in the bucket until it is filled with c_1 keys. A later insertion attempting to fall in the same bucket will force it to be stretched into a bucket of the next size c_2, and so on. We call the stretching of a bucket to become a larger bucket a *conversion*, and because of this stretching, some authors call such stretchable buckets "*elastic*." Thus types $1, \ldots, p-1$ are pointerless, but type p must contain m pointers. Notice that the inequalities (1.1) are all strict except possibly the last. For it is conceivable that $c_{p-1} < c_p$, in which case an insertion in a node of type $p-1$ converts it into a partially filled node of type p that will continue to accept some more keys until it is filled before the keys are sent to the subtrees. On the other hand, it is conceivable that in some memory management schemes $c_{p-1} = c_p = m - 1$, in which case an insertion in the pointerless type $p-1$ will convert it into a type p bucket with the same number of keys $c_p = m - 1$ but with an extra m pointers; the new key will then appear by itself in a new node (of type 1, of course) that will be linked as a subtree. Figure 1 illustrates the growth of a tree with branch factor 8 from eight keys under a scheme that allows buckets of capacities $c_1 = 3$, $c_2 = 5$, $c_3 = 7$ (and type 3 has eight pointers).

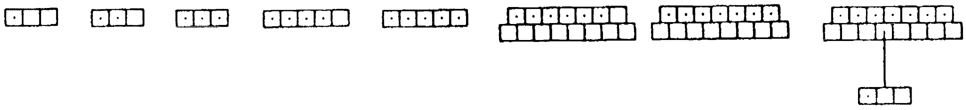


FIG. 1. *The step-by-step growth of an 8-way tree with node capacities* 3, 5, *and* 7.

The advantage of random search trees is that they are almost complete (i.e., shortest possible) with very high probability (see [20], [4], [17], [18], [22]). Thus time-performance is almost the best possible. Under a *fixed allocation scheme* (a scheme with only one type node, i.e., $p = 1$), search trees are known to be wasteful of space (see [12], [1], [21], and [17]; [12, Exercise 6.3.20, p. 501] presents the first known analysis of trees with buckets). We investigate in this paper the possibility of rectifying the situation via elastic buckets so as to get good space performance too. The notion of increasing sizes for different type nodes of a linked data structure was introduced before in the class of balanced trees. Frederickson [8] and Lomet [15] used elastic buckets to implement increasing size nodes when they overflow instead of the usual splitting of nodes in B-trees and their derivatives. Baeza-Yates and Larson [2] analyzed mathematically the storage utilization of B^+-trees with elastic buckets. The possibility of using elastic buckets was also made implicit by Hoshi and Flajolet [10], who computed the average number of nodes of each type in a paged quadtree. Baeza-Yates [1] found the optimal second size in the case where two page sizes are allowed for a multiway search tree.

In practice, an algorithm on the m-ary tree may require knowledge of the sizes of the nodes and most memory management systems reserve some bytes in each block for system use. These hidden bytes may hold information like the size of the bucket. We shall call these hidden bytes the *hidden tag*. In most memory allocation schemes, the hidden tag has a uniform length in all buckets regardless of their size. Practical multitype memory management systems include the following:

(a) The *exact fit scheme* where type i buckets contain the exact space for i keys and the hidden tag, $i = 1, \dots, p-1$, and type p contains the exact space for $m-1$ keys, m pointers, and the hidden tag.

(b) The *buddy system scheme*, which allocates space in blocks that are proper powers of two words. One word is reserved in each bucket for a hidden tag. Thus in an application with real or integer data (where each datum needs one word of memory), a bucket of size 2^k words can accommodate $2^k - 1$ data items (even if the data are not numeric, uniform pointers occupying one word each may be used for indirect addressing). Thus the bucket capacities (in terms of keys) are $1, 3, 7, 15, \dots, 2^p - 1$ (with the branching factor $m = 2^p$, a proper power of 2). This is the choice method of accessing the free space in some popular systems such as the simulation programming language SIMSCRIPT and the MALLOC memory allocation function in the programming language C (on the UNIX 4.3 BSD operating system).

Let $X_n^{(i)}$ be the number of nodes of type i after n insertions. We shall investigate the multivariate structure of the tree by examining the vector $\mathbf{X}_n = (X_n^{(1)}, \dots, X_n^{(p)})^T$. We shall compute the asymptotic average of this vector. We shall further find the covariance matrix and (for a range of m) the joint asymptotic distribution of the components of $\mathbf{X}_n$. It will turn out that in practical systems all variances and covariances are linear in n for $3 \le m \le 26$, and then they suffer from a phase transition: For $m > 26$, the variances and covariances become a superlinear (but subquadratic) function of n. The asymptotic distribution for the linear range of the variances and covariances ($3 \le m \le 26$) is multivariate normal (multinormal).

An associated multivariate problem is a study of the number of conversions between types. The total number of conversions may not be an accurate measure of the cost of conversions since different conversions involve movements of different block sizes of data. A more accurate measure is obtained by studying the number of conversions of each type. More precisely, let $C_n^{(i)}$ be the number of times a node of type $i-1$ is converted into a node of type i (a conversion of type i), $i = 1, \dots, p$, and $C_n^{(1)}$ be interpreted as the number of times a node is created for the first time, i.e., the number of times a **NIL** pointer (a node of type 0) is converted into a node of type 1. We are interested in the behavior of the vector

$$\mathbf{C}_n = (C_n^{(1)}, \dots, C_n^{(p)})^T.$$

The same questions about average, covariance matrix, and any phase transitions that may appear therein, and limiting joint distribution may be addressed again. Results about $\mathbf{C}_n$ may be found from the behavior of $\mathbf{X}_n$ since their components are related as follows. When a bucket of capacity c_i first appears, it increases $C_n^{(i)}$ by one. But this node may grow into a bigger bucket later on. In other words, in the final tree, every node of capacity c_i or larger has passed at some point in its history through one conversion of the ith type or

$$C_n^{(i)} = X_n^{(i)} + X_n^{(i+1)} + \cdots + X_n^{(p)},$$

or $\mathbf{C}_n = \mathbf{J}_p \mathbf{X}_n$, in matrix form, where $\mathbf{J}_p$ is the $p \times p$ upper diagonal matrix whose upper diagonal elements are all 1. To avoid duplicating the effort to find results about each bucket type, we shall instead study the linear combination

$$\mathcal{X}_n = a_1 X_n^{(1)} + \cdots + a_p X_n^{(p)}$$

for any arbitrary real numbers $a_1, \dots, a_p$. This adds flexibility to our calculations: results concerning the mean and variance of an individual type, type i, say, are obtained by setting $a_j = 0$, $j \ne i$, and $a_i = 1$. For the covariance between types i and j we set $a_i = a_j = 1$ and $a_s = 0$, $s \notin \{i, j\}$, to obtain

$$\begin{aligned}\mathbf{Var}[\mathcal{X}_n] &= \mathbf{Var}[X_n^{(i)} + X_n^{(j)}] \\ &= \mathbf{Var}[X_n^{(i)}] + \mathbf{Var}[X_n^{(j)}] + 2\mathbf{Cov}[X_n^{(i)}, X_n^{(j)}].\end{aligned}$$

Results about the total number of nodes allocated are obtained by setting $a_1 = a_2 = \cdots = a_p = 1$, and results about the actual number of words allocated may be obtained by setting a_i to the actual number of words taken by a bucket of type i (e.g., $a_i = k_1 i + k_2$ for the exact fit scheme with constants k_1 and k_2, and $a_i = 2^i$ under the buddy system scheme). The added advantage of studying the linear combination $\mathcal{X}_n$ is that it prepares for use of the Cramér-Wold device to prove the joint asymptotic normality of the components of the vector $\big(X_n^{(1)}, \dots, X_n^{(p)}\big)^T$ in a range of m.

We shall use the following notation for the falling and rising factorials:

$$(z)_j \overset{\text{def}}{=} z(z-1)\cdots(z-j+1), \qquad (z)_0 \overset{\text{def}}{=} 1,$$

$$\langle z\rangle_j \overset{\text{def}}{=} z(z+1)\cdots(z+j-1), \qquad \langle z\rangle_0 \overset{\text{def}}{=} 1.$$

We shall also denote by $[z^n]f(z)$ the coefficient of z^n in $f(z)$. The kth harmonic number $1 + 1/2 + 1/3 + \cdots + 1/k$ will be denoted by H_k. The rest of the paper will be organized into sections as follows: In §2, we set up a partial differential equation for a supergenerating function that generates the moment generating functions of the linear combination $\mathcal{X}_n$ for all values of n. Although this partial differential equation is not tractable, we can develop from it tractable ordinary differential equations for the moments. In §3, we solve the ordinary differential equation for the first moment and obtain an asymptotic estimate for the components of $\mathcal{X}_n$, and in §4, we solve for the second moment and obtain the asymptotic covariance matrix between the types. The main result of this paper, the asymptotic joint normality of the components of $\mathcal{X}_n$ in a range of m, is presented in §5. In §6, we use the fixed bucket, exact fit, and the buddy system schemes as illustrating examples and provide comparisons between these practical methods. Section 7 is a conclusion.

2. A differential equation for a super moment generating function of $\mathcal{X}_n$. Given that the number of keys that end up in the jth subtree is n_j ($n_1 + \cdots + n_m = n - m + 1$), the ith type bucket satisfies the recurrence

$$X_n^{(i)} = \begin{cases} X_{n_1}^{(i)} + X_{n_2}^{(i)} + \cdots + X_{n_m}^{(i)}, & i = 1, 2, \dots, p-1, \\ X_{n_1}^{(i)} + X_{n_2}^{(i)} + \cdots + X_{n_m}^{(i)} + 1, & i = p, \end{cases}$$

valid for $n \ge c_p + 1 = m$. Thus, given $n_1, \dots, n_m, n \ge m$,

(2.1)

$$\begin{aligned}\mathcal{X}_n &= a_1[X_{n_1}^{(1)} + \cdots + X_{n_m}^{(1)}] + a_2[X_{n_1}^{(2)} + \cdots + X_{n_m}^{(2)}] + \cdots + a_p[X_{n_1}^{(p)} + \cdots + X_{n_m}^{(p)} + 1] \\ &= a_p + \sum_{j=1}^{m}\big(a_1 X_{n_j}^{(1)} + a_2 X_{n_j}^{(2)} + \cdots + a_p X_{n_j}^{(p)}\big) \\ &= \mathcal{X}_{n_1} + \mathcal{X}_{n_2} + \cdots + \mathcal{X}_{n_m} + a_p.\end{aligned}$$

Let $\phi_n(t)$ be the moment generating function of $\mathcal{X}_n$, and denote the number of keys in the jth subtree by τ_j, $j = 1, \dots, m$. We can write a recurrence for $\phi_n(t)$ by conditioning as follows

$$\begin{aligned}\phi_n(t) &= \mathbf{E}\big[\exp(\mathcal{X}_n t)\big]\\ &= \sum_{n_1+\cdots+n_m=n-m+1} \mathbf{E}\big[\exp(\mathcal{X}_n t) \mid \tau_1 = n_1, \dots, \tau_m = n_m\big]\\ &\qquad\qquad \times \Pr\big(\tau_1 = n_1, \dots, \tau_m = n_m\big).\end{aligned}$$

The condition is known to occur with probability $\binom{n}{m-1}^{-1}$ under the random permutation model (see [19] for a probabilistic proof or [16] for a combinatorial proof). So from the recurrence (2.1),

$$\phi_n(t) = \frac{1}{\binom{n}{m-1}} \sum_{n_1+\cdots+n_m=n-m+1} \mathbf{E}\big[\exp\{(\mathcal{X}_{n_1} + \cdots + \mathcal{X}_{n_m} + a_p)t\}\big]$$

or

$$(2.2)\qquad (n)_{m-1}\phi_n(t) = (m-1)!\, \exp\big(a_p t\big) \sum_{n_1+\cdots+n_m=n-m+1} \mathbf{E}\big[\exp(\mathcal{X}_{n_1} t)\big] \cdots \mathbf{E}\big[\exp(\mathcal{X}_{n_m} t)\big]$$

(valid only for $n \geq m$), where the last relation follows from the independence of the structures of the subtrees (after checking the root, the insertion algorithm goes into a subtree and does not look at any of the other subtrees after that point). Define the supergenerating function

$$\Phi(x, t) = \sum_{n=0}^{\infty} \phi_n(t) x^n.$$

Multiply both sides of (2.2) by x^{n-m+1} and sum over $n \geq m$ to get

$$(2.3)\qquad \frac{\partial^{m-1}\Phi(x,t)}{\partial x^{m-1}} = (m-1)!\, \exp\big(a_p t\big)\Phi^m(x,t) + (m-1)!\,\big[\phi_{m-1}(t) - \exp\big(a_p t\big)\big].$$

The last term on the right appears owing to the necessary corrections for $n = 0, 1, \dots, m-1$ as the general recurrence (2.1) is not valid for these values of n.

The partial differential equation (2.3) seems to be intractable; however, tractable ordinary differential equations for the moments may be obtained from it as in several instances discussed in a few papers and reported in [17]. Derivatives of $\Phi(x, t)$ with respect to t at $t = 0$ are generating functions for the moments of $\mathcal{X}_n$. Calling the kth derivative $G_k(x)$, we can easily see that

$$G_k(x) = \frac{\partial^k \Phi(x, 0)}{\partial t^k} = \sum_{n=0}^{\infty} \mathbf{E}[\mathcal{X}_n^k] x^n$$

is the ordinary generating function for the kth moment of $\mathcal{X}_n$.

3. The first moment. In particular, $G_1(x)$ generates the sequence of averages $\mathbf{E}\big[\mathcal{X}_n\big]$, $n \geq 0$. Differentiating both sides of (2.3) once with respect to t and evaluating it at $t = 0$ we obtain

(3.1)

$$\frac{d^{m-1}}{dx^{m-1}} G_1(x) = m!\, G_1(x)\Phi^{m-1}(x,0) + (m-1)!\, a_p \Phi^m(x,0) + (m-1)!\,\big[\phi'_{m-1}(0) - a_p\big].$$

But $\Phi(x,0) = \sum_{n=0}^{\infty} x^n = (1-x)^{-1}$ for $|x| < 1$ and

$$\begin{aligned}\phi'_{m-1}(0) &= \frac{d}{dt}\big(\mathbf{E}[\exp(t\mathcal{X}_{m-1})]\big)_{t=0}\\ &= \mathbf{E}\big[a_1 X_{m-1}^{(1)} + \cdots + a_{p-1}X_{m-1}^{(p-1)} + a_p X_{m-1}^{(p)}\big].\end{aligned}$$

From the behavior of the algorithm with multitype buckets

$$X_{m-1}^{(i)} \equiv 0, \qquad i = 1, \ldots, p-2,$$

and

$$X_{m-1}^{(p-1)} \equiv \delta_{c_p, c_{p-1}}, \qquad X_{m-1}^{(p)} \equiv 1 - \delta_{c_p, c_{p-1}},$$

where δ is the Kronecker delta function. Hence (3.1) can be written as

$$\frac{d^{m-1}}{dx^{m-1}} G_1(x) = \frac{m!\, G_1(x)}{(1-x)^{m-1}} + H_1(x), \tag{3.2}$$

where

$$H_1(x) = \frac{(m-1)!\, a_p}{(1-x)^m} + (m-1)!\,(a_{p-1} - a_p)\delta_{c_p, c_{p-1}}.$$

Equation (3.2) can be solved under the obvious initial conditions

$$\frac{d^i}{dx^i} G_1(0) = i!\, \mathbf{E}[\mathcal{X}_i] = i!\, a_r, \qquad i = 0, 1, \ldots, m-2, \tag{3.3}$$

where by definition: (i) $c_0 = 0$, (ii) $r = 0$ and $a_0 = 0$ for $i = 0$, and (iii) r is defined by $c_{r-1} < i \le c_r$ for $i = 1, \ldots, m-2$ (i.e., r is the index of the type that corresponds to the smallest capacity for i keys).

The homogeneous part of the differential equation (3.2) is of the Euler type and has solution $(1-x)^{-\lambda_j}$ for each λ_j that is a solution to the characteristic equation

$$\psi(\lambda) = 0,$$

where

$$\psi(\lambda) = \psi_m(\lambda) \overset{\text{def}}{=} \lambda(\lambda+1)\cdots(\lambda+m-2) - m!.$$

Mahmoud [17] discusses in detail the role of the roots of this characteristic function for a class of random variables called *additive* on m-ary trees and surveys the results of several papers solving other instances. Our linear combination $\mathcal{X}_n$ falls into this class.

We list here some of the properties of the roots of the characteristic function, $\psi(\lambda)$, for the sake of completeness:

(a) $\psi(2) = 0$; denote this root by λ_1; λ_1 is a simple root and $\Re\lambda_j < 2$, $j = 2, \ldots, m-1$. The number $-m$ is a simple root if and only if m is odd. All the other roots of $\psi(\lambda)$ are complex valued, each of multiplicity one.

(b) No two roots may have the same real part unless they are mutually conjugate.

(c) For $m \ge 4$ there is precisely one pair of complex conjugate roots $\lambda_2 = \alpha + i\beta$, $\lambda_3 = \alpha - i\beta$ ($\beta > 0$), with $\alpha < 2$ being the second largest among the real parts of the roots. Thus, $\Re\lambda_j < \alpha$ for $4 \le j \le m-1$.

(d) $\alpha = \alpha_m \to 2$ as $m \to \infty$. Moreover, $\alpha < 1$ for $3 \le m \le 13$ and $1 < \alpha < 3/2$ for $14 \le m \le 26$.

(e) $\psi'(\lambda_k) = m!\sum_{j=0}^{m-2}(\lambda_k + j)^{-1}$, $1 \le k \le m-1$. Another useful expression for $\psi'(\lambda_k)$ is

$$\prod_{\substack{1\le j\le m-1\\ j\ne k}} (\lambda_k - \lambda_j).$$

For more details on the properties of the roots of $\psi(\lambda)$, see [21].

It can be checked that the term $H_1(x)$ in (3.2) gives rise to the particular solution

$$h_1(x) = -\frac{a_p}{(m-1)(1-x)} - \frac{\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)}{m+(-1)^m}(1-x)^{m-1}.$$

Thus the complete solution to (3.2) is given by

$$G_1(x) = \sum_{j=1}^{m-1} A_j(1-x)^{-\lambda_j} + h_1(x), \tag{3.4}$$

where A_j are the constants of integration obtained from (3.3) by solving the linear system of equations

$$\sum_{j=1}^{m-1}\langle\lambda_j\rangle_i A_j = \frac{d^i}{dx^i}G_1(0) - \frac{d^i}{dx^i}h_1(0), \qquad i = 0, \dots, m-2,$$

which can be written in matrix form as

$$\mathbf{V}_{m-1}\mathbf{A} = \mathbf{b},$$

where

$$\begin{aligned}\mathbf{V}_{m-1} &= \mathbf{V}_{m-1}(\lambda_1, \dots, \lambda_{m-1})\\ &= \left[\langle\lambda_j\rangle_i\right], \qquad i = 0, \dots, m-2, \quad j = 1, \dots, m-1.\end{aligned}$$

After some elementary row operations the determinant of $\mathbf{V}_{m-1}$ reduces to the well-known Van Dermonde determinant of order $m-1$. In the vector $\mathbf{A}= \left(A_1, \dots, A_{m-1}\right)^T$, each of the constants A_j, $j = 1, \dots, m-1$ is of course a function of $a_1, \dots, a_p$. For each allocation scheme and each choice of the numbers $a_1, \dots, a_p$, the constants in the vector $\mathbf{b} = \left(b_0, \dots, b_{m-2}\right)^T$ are completely known from (3.3) to be

$$b_i = \begin{cases} -h_1(0), & i = 0,\\ i!\,a_r - \dfrac{d^i}{dx^i}h_1(0), & i = 1, \dots, m-2,\end{cases}$$

where r is defined by $c_{r-1} < i \le c_r$ for $i = 1, \dots, m-2$. Specific examples of bucket multityping will be used to illustrate the procedure of obtaining the constants A_j at the end of the paper. Thus for any given allocation scheme, we have a procedure for obtaining the constants A_j, $j = 1, \dots, m-1$. Extracting the coefficients of x^n from both sides of (3.4) we obtain

$$\mathbf{E}[\mathcal{X}_n] = \sum_{j=1}^{m-1}[x^n]A_j(1-x)^{-\lambda_j} - \frac{a_p}{m-1} - \frac{\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)}{m+(-1)^m}(-1)^n\binom{m-1}{n}.$$

According to the binomial theorem, for any complex number λ, the imaginary part of the coefficient of x^n will only provide oscillation with no growth in magnitude; the growth comes only from the real part, and $n^{\Re\lambda-1}$ is its exact order of magnitude as $n \to \infty$. Hence the root with the largest real part, $(\lambda = 2)$ of the characteristic equation gives a term that asymptotically dominates all other terms, and the average of $\mathcal{X}_n$ can be written in the form

$$\begin{aligned}\mathbf{E}[\mathcal{X}_n] &= A_1[x^n](1-x)^{-2} + [x^n]\left(\sum_{j=2}^{m-1}A_j(1-x)^{-\lambda_j}\right) - \frac{a_p}{m-1}\\ &\quad - \frac{\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)}{m+(-1)^m}(-1)^n\binom{m-1}{n}\\ &= A_1 n + \left(A_1 - \frac{a_p}{m-1}\right) + O\left(n^{\alpha-1}\right)\\ &= A_1 n + O\left(n^b\right),\end{aligned}$$

where $b = \max\{0, \alpha - 1\}$. From the list of properties of the roots of the characteristic equations for $m \leq 13$ and $\alpha \leq 1$ the lower-order term is $O\left(1\right)$ and for $m > 13$ it grows sublinearly with n.

4. The covariance between types. Differentiating (2.3) twice with respect to t at $t = 0$ gives

$$\frac{d^{m-1}}{dx^{m-1}}G_2(x) = \frac{m!\,G_2(x)}{(1-x)^{m-1}} + H_2(x),$$

where

$$\begin{aligned}H_2(x) &\overset{\text{def}}{=} \frac{m!\,(m-1)G_1^2(x)}{(1-x)^{m-2}} + \frac{(m-1)!\,a_p^2}{(1-x)^m} + \frac{2a_p(m!)G_1(x)}{(1-x)^{m-1}}\\ &\quad + (m-1)!\,(a_{p-1}^2 - a_p^2)\delta_{c_p,c_{p-1}},\end{aligned}$$

and $G_1(x)$ was completely determined in the previous section. Thus, in a manner similar to the integration steps used for $G_1(x)$, we obtain

$$G_2(x) = \sum_{j=1}^{m-1}K_j(1-x)^{-\lambda_j} + h_2(x),$$

where K_j are integration constants, λ_j are the roots of the characteristic equation for $j = 1, \ldots, m-1$, and $h_2(x)$ is the particular solution corresponding to the integration of $H_2(x)$. The initial conditions are

$$(4.1)\qquad \frac{d^i}{dx^i}G_2(0) = i!\,\mathbf{E}[\mathcal{X}_i^2] = i!\,a_r^2, \qquad i = 0, 1, \ldots, m-2,$$

where by definition: (i) $c_0 = 0$, (ii) $r = 0$ and $a_0 = 0$ for $i = 0$, and (iii) r is defined by $c_{r-1} < i \leq c_r$ for $i = 1, \ldots, m-2$ (i.e., r is the index of the type that corresponds to the smallest capacity for i keys). Thus the constants K_j are solutions of a linear system that may be written in the form

$$\mathbf{V}_{m-1}\mathbf{K} = \mathbf{f}.$$

Note that like A_j, the constants K_j are functions of $a_1, \ldots, a_p$. For each allocation scheme and each choice of the numbers $a_1, \ldots, a_p$, the constants in the vector $\mathbf{f} = \big(f_0, \ldots, f_{m-2}\big)^T$ are completely known from (4.1) to be

$$f_i = \begin{cases} -h_2(0), & i = 0, \\ i!\,a_r^2 - \dfrac{d^i}{dx^i}h_2(0), & i = 1, \ldots, m-2, \end{cases}$$

where r is defined by $c_{r-1} < i \le c_r$ for $i = 1, \ldots, m-2$. Specific examples of bucket multityping will be used to illustrate the procedure of obtaining the constants K_j at the end of the paper.

Observe that all the terms of $H_2(x)$ have the general form $c(1-x)^{-\lambda-m+1}$. As in [17], each term of $H_2(x)$ of the form $c(1-x)^{-\lambda-m+1}$ contributes to the particular solution $h_2(x)$ a term of the form

$$\frac{c}{\psi(\lambda)}(1-x)^{-\lambda},$$

provided that $\psi(\lambda) \ne 0$. If $\psi(\lambda) = 0$, then a term of the form

$$\frac{c}{\psi'(\lambda)}(1-x)^{-\lambda}\ln\Big(\frac{1}{1-x}\Big)$$

is contributed to the particular solution. Thus, $h_2(x)$ can be obtained by organizing the terms of $H_2(x)$ into cases according to the powers λ in the term $(1-x)^{-\lambda-m+1}$. The details of this calculation are given in the appendix. The final conclusion is the following:

THEOREM 1.

$$\mathbf{Var}\big[\mathcal{X}_n\big] = \gamma_m n + c_m(n)n^{2\alpha-2} + O\big(n^b\big)$$

for every $b > \max\{0, 2\alpha-2, \alpha-1\}$, *where* $\gamma_m = K_1 + A_1^2 + \frac{2a_pA_1}{m-1}$ *and* $c_m(n)$ *is the* $\frac{\pi}{\beta}$*-periodic function*

$$c_m(n) = \sum_{2\le j,k\le 3} A_jA_k\Big[\frac{m!\,(m-1)}{\psi(\lambda_{jk})\Gamma(\lambda_{jk})} - \frac{1}{\Gamma(\lambda_j)\Gamma(\lambda_k)}\Big]n^{i\Im\lambda_{jk}}. \tag{4.2}$$

The factors A_j, K_j, $j = 1, \ldots, m-1$, depend on the allocation scheme and on our choice of $a_1, \ldots, a_p$. In the computation of variances and covariances, we shall need the values of A_j and K_j corresponding to setting all a_i's to 0 except for a pair of indices, say, s and t (not necessarily distinct). We denote by $A_j^{(st)}$ and $K_j^{(st)}$ the values of A_j and K_j with $a_r = 0, r \notin \{s, t\}$, and $a_s = a_t = 1$. Similarly, $\gamma_m^{(ij)}$ and $c_m^{(ij)}(n)$ refer to γ_m and $c_m(n)$ when $a_i = a_j = 1$ and all the other a's are 0. For the variances of the individual types we compute

$$\begin{aligned}\mathbf{Var}\big[X_n^{(i)}\big] &= \mathbf{Var}\big[\mathcal{X}_n\big]_{\substack{a_i=1\\ a_k=0;\, k\ne i}} \\ &= \gamma_m^{(ii)}n + c_m^{(ii)}(n)\,n^{2\alpha-2} + O\big(n^b\big),\end{aligned}$$

for every $b > \max\{0, \alpha-1, 2\alpha-2\}$. Thus, the variance of each bucket type satisfies a formula that is essentially like that of $\mathbf{Var}\big[\mathcal{X}_n\big]$. For practical memory management systems, both $\gamma_m^{(ii)}(n)$ and $c_m^{(ii)}(n)$ are nonzero as illustrated by a few examples at the end of the paper. That is, for such memory management systems, the variance of each bucket type suffers from a phase transition at $m = 26$. The value 26 is an exact result based on an analysis of the roots of the characteristic equation $\psi(\lambda) = 0$ mentioned in §3 and is independent of the bucket sizes. Furthermore, the covariances between different types of buckets may be found from the relation

$$\begin{aligned}\mathbf{Var}[\mathcal{X}_n] \underset{\substack{a_i=a_j=1\\ a_s=0;\ s\notin\{i,j\}}}{} &= \mathbf{Var}\big[X_n^{(i)} + X_n^{(j)}\big]\\ &= \mathbf{Var}\big[X_n^{(i)}\big] + \mathbf{Var}\big[X_n^{(j)}\big] + 2\mathbf{Cov}\big[X_n^{(i)}, X_n^{(j)}\big]\end{aligned}$$

or

$$\begin{aligned}\mathbf{Cov}\big[X_n^{(i)}, X_n^{(j)}\big] &= \frac{1}{2}\big[\gamma_m^{(ij)} - \gamma_m^{(ii)} - \gamma_m^{(jj)}\big]n\\ &\quad + \frac{1}{2}\big[c_m^{(ij)}(n) - c_m^{(ii)}(n) - c_m^{(jj)}(n)\big]n^{2\alpha-2} + O\big(n^b\big)\end{aligned}$$

for every $b > \max\{0, \alpha - 1, 2\alpha - 2\}$. For practical memory management systems, the covariances also have phase transitions after branch factor 26, from a linear behavior to a superlinear (but subquadratic) behavior with oscillations. It appears that the phase transition persists in all practical memory management systems. This phase transition is known to be true for trees with a fixed bucket size implementation and the study of multitype buckets could be thought of as lumping nodes holding a certain number of keys into one type. It is therefore expected that properties such as phase transition in the variance for trees with a fixed bucket capacity will carry over to multitype search trees. Since the asymptotic variances and covariances between the components of $\mathbf{X}_n = \big(X_n^{(1)}, \dots, X_n^{(p)}\big)^T$ all have the same linear order of magnitude for $3 \le m \le 26$, the asymptotic variance-covariance matrix is of the form $\mathbf{\Lambda}_p n$, with $\mathbf{\Lambda}_p$ being a $p \times p$ matrix of coefficients for this range of m.

5. Asymptotic joint normality of the different types. Let $\hat{X}_n^{(i)}$ be the normalized random variable $\big(X_n^{(i)} - \mathbf{E}\big[X_n^{(i)}\big]\big)/\sqrt{n}$. The main result of this paper is the following:

THEOREM 2. *In the range of m for which $\alpha < 3/2$ (which happens if $3 \le m \le 26$), the components of the vector $(\hat{X}_n^{(1)}, \dots, \hat{X}_n^{(p)})^T$ are asymptotically jointly multinormal with a covariance structure $\mathbf{\Lambda}_p$ that follows from the procedure of the previous section.*

The idea of the proof is similar to that in the case of the fixed allocation scheme [21] but extends their previous result to cover the entire range of linearity of the variance, where they only showed asymptotic normality for $3 \le m \le 15$. For the univariate random variable $\mathcal{X}_n$, we give an outline of a streamlined proof by only listing the adjustments needed in the proof of the fixed allocation scheme by a few lemmas without proof except for Lemma 4 in the following text. A proof for Lemma 4 is given to show the extension to the entire range of linearity of variances and covariances. The idea in the proof of asymptotic normality is to compare the moment generating function $\phi_n(t)$ (for which the recursive nature of the data partitioning process provides a recurrence) with the moment generating function

$$\xi_n(t) = \exp\left\{\mathbf{E}[\mathcal{X}_n]t + \mathbf{Var}[\mathcal{X}_n]\frac{t^2}{2}\right\},$$

which is the moment generating function of a normally distributed random variable with mean and variance like $\mathcal{X}_n$. The proof works by conditioning on the number of keys in the subtrees: for any particular partitioning $(n_1, \dots, n_m)$ of keys into the subtrees (of course $n_1 + \cdots + n_m =$

$n-m+1$), $\phi_n(t)$ can be written as an expression involving the product $\phi_{n_1}(t)\ldots\phi_{n_m}(t)$ and $\xi_n(t)$ can be written as an expression involving the product $\xi_{n_1}(t)\ldots\xi_{n_m}(t)$. The similarity of the two forms is advantageously used in an inductive proof to give tight upper and lower bounds on $\phi_n(t)/\xi_n(t)$. In what follows, $\mathbf{n}$ denotes a row vector $(n_1,\ldots,n_m)$ whose components satisfy $n_1+\cdots+n_m=n-m+1$.

LEMMA 1.

$$\phi_n(t)=\frac{\exp(a_pt)}{\binom{n}{m-1}}\sum_{\mathbf{n}}\prod_{j=1}^{m}\phi_{n_j}(t).$$

LEMMA 2.

$$\frac{\exp(a_pt)}{\binom{n}{m-1}\xi_n(t)}\sum_{\mathbf{n}}\prod_{j=1}^{m}\xi_{n_j}(t)=1+\Theta\big(n^c\big)$$

with

$$c=\max\{3c_1,c_1+c_2,2c_2\},$$

$$c_1=\max\{-1/2,\alpha-3/2\},\quad \textit{and}\quad c_2=\max\{2\alpha-3,-1,\alpha-2\}.$$

LEMMA 3. *For large n and every* $0\le\nu\le n$:

$$\exp\Big(-\frac{d\nu n^c}{m-1}\Big)\le\frac{\phi_\nu(t)}{\xi_\nu(t)}\le\exp\Big(\frac{d\nu n^c}{m-1}\Big)$$

for some positive number d.

LEMMA 4.

$$\lim_{n\to\infty}\frac{\phi_n(t)}{\xi_n(t)}=1.$$

Proof. First, observe that if $m\le 26$, then $\alpha<3/2$, and the constant c can be computed from Lemma 2 yielding $c<-0.00255$. Choose $0<\delta<-c$. Let $\nu(n)=\lfloor n^\delta\rfloor$, and note that $\nu(n)$ ranges over the set of positive integers $I\!\!N$ (with repetitions) as n varies from 1 to ∞. Conversely, for each ν, we can define $n_1=n_1(\nu)=\max\big\{n\big|\lfloor n^\delta\rfloor=\nu\big\}$. Observe that $\nu<n_1$ and $n_1=O(\nu^{1/\delta})$, as $\nu\to\infty$ and we can apply Lemma 3 with large ν to get

$$\exp\Big(-\frac{d\nu n_1^c}{m-1}\Big)\le\frac{\phi_\nu(t)}{\xi_\nu(t)}\le\exp\Big(\frac{d\nu n_1^c}{m-1}\Big)$$

or

$$\exp\Big(-O\big(\nu^{(c+\delta)/\delta}\big)\Big)\le\frac{\phi_\nu(t)}{\xi_\nu(t)}\le\exp\Big(O\big(\nu^{(c+\delta)/\delta}\big)\Big).$$

According to our choices of δ, the right and left ends of the above inequality tend to 1 as $\nu\to\infty$, giving us the desired result. □

LEMMA 5. $(\mathcal{X}_n - \mathbf{E}[\mathcal{X}_n])/\sqrt{\mathbf{Var}[\mathcal{X}_n]}$ *converges in distribution to the standard normal distribution with mean* 0 *and variance* 1.

Since

$$\frac{\mathcal{X}_n - \mathbf{E}[\mathcal{X}_n]}{\sqrt{\mathbf{Var}[\mathcal{X}_n]}} \xrightarrow{\mathcal{D}} N(0, 1),$$

this implies that

$$\frac{\mathcal{X}_n - \mathbf{E}[\mathcal{X}_n]}{\sqrt{n}} \xrightarrow{\mathcal{D}} N\left(0, \lim_{n\to\infty} \frac{\mathbf{Var}[\mathcal{X}_n]}{n}\right),$$

which implies that

$$\frac{\mathcal{X}_n - \mathbf{E}[\mathcal{X}_n]}{\sqrt{n}} \xrightarrow{\mathcal{D}} N\left(0, \sum_{1\le i,j\le p} a_i a_j s_{ij}\right)$$

where s_{ij} is the (i, j)th entry of $\mathbf{\Lambda}_p$ or

$$\frac{\mathcal{X}_n - \mathbf{E}[\mathcal{X}_n]}{\sqrt{n}} \xrightarrow{\mathcal{D}} N(0, \mathbf{a}^T \mathbf{\Lambda}_p \mathbf{a}).$$

However, $N(0, \mathbf{a}^T\mathbf{\Lambda}_p\mathbf{a})$ is the distribution of $\mathbf{a}^T\mathbf{Y}$ where $\mathbf{Y}$ is a multinormal vector of p components and variance-covariance matrix $\mathbf{\Lambda}_p$. So, this can be summed up as

$$\mathbf{a}^T\left(\frac{\mathbf{X}_n - \mathbf{E}[\mathbf{X}_n]}{\sqrt{n}}\right) \xrightarrow{\mathcal{D}} \mathbf{a}^T\mathbf{Y}$$

for all p-component row vector $\mathbf{a}^T \in I\!R^p$ and Theorem 2 follows from the Cramér-Wold device (see [3]).

COROLLARY 1. *For* $3 \le m \le 26$, *the normalized number of buckets of type* i, *that is* $\hat{\mathcal{X}}_n^{(i)}$, *converges in distribution to the normal distribution with mean* 0 *and variance* $\gamma_m^{(ii)}$.

As discussed in the introduction, each $C_n^{(i)}$ (the number of conversions from type $i - 1$ bucket to type i bucket) is a linear combination of $\mathcal{X}_n^{(1)}, \ldots, \mathcal{X}_n^{(p)}$. In matrix form, we have

$$\mathbf{C}_n = \mathbf{J}_p\mathbf{X}_n.$$

Whence

$$\mathbf{Cov}[\mathbf{C}_n] = \mathbf{J}_p\mathbf{X}_n\mathbf{J}_p^T.$$

As is well known, linear combinations of normal variates are themselves normal, and we have

COROLLARY 2. *Let* $\hat{C}_n^{(i)} = (C_n^{(i)} - \mathbf{E}[C_n^{(i)}])/\sqrt{n}$. *For* $3 \le m \le 26$, *the random vector* $(\hat{C}_n^{(1)}, \ldots, \hat{C}_n^{(p)})^T$ *is asymptotically jointly normal (with mean* $\mathbf{0}$ *and covariance matrix* $\mathbf{J}_p\mathbf{\Lambda}_p\mathbf{J}_p^T$*) and each* $\hat{C}_n^{(i)}$ *converges in distribution to the normal distribution with mean* 0 *and variance equal to the* (i, i)th *entry of* $\mathbf{J}_p\mathbf{\Lambda}_p\mathbf{J}_p^T$.

6. Some illustrating examples. (a) *Fixed Bucket Scheme.* Under several operating systems, like IBM's OS/2 for personal computers and IBM's VM/CMS, virtual memory is used where external storage devices can be thought of as extensions to the primary computer memory. Communication with the external devices takes place in *fixed* units called pages (a typical page size is: 0.5 – 4K bytes where 1K = 1024 bytes). For these operating systems, a tree node with all its keys and pointers may be designed as one page giving rise to a branch factor in the range of hundreds. This special case corresponds to $p = 1$, as we have only one type of nodes with capacity $c_1 = m - 1$. Let S_n be the total actual space in words taken by the tree. If each bucket (page) size is θ words, then

$$\mathbf{E}[S_n] = \mathbf{E}[\theta \mathcal{X}_n]_{a_1=1} = \frac{\theta n}{2(H_m - 1)} + O(n^b)$$

with $b = \max\{0, \alpha - 1\} < 1$. This result was derived by different methods in [12], [1], and [21]. Moreover, the result about the variance and its phase transition and the asymptotic normality of S_n appears in [21]. These results are special cases of the results in this paper as $S_n = \mathcal{X}_n|_{a_1=1}$. As a consequence of a forthcoming discussion about the exact fit scheme, it can be shown that $Y_n^{(i)}$, the number of fixed buckets with i keys, $i = 1, \dots, m-1$, is asymptotically normal and its variance suffers from a phase transition after $m = 26$. Also, $Y_n^{(1)}, \dots, Y_n^{(m-1)}$ are asymptotically jointly normal with covariances suffering from phase transitions after $m = 26$. The space utilization of the fixed bucket will be discussed in the exact fit scheme method.

(b) *Exact Fit Scheme.* This is a multitype scheme with complete control over memory. A request for a certain number of bytes is honored exactly. This kind of control may be achieved by accessing a "heap" of free space. When a tree grows under this scheme, the type i bucket for $i = 1, \dots, p-1$ will have the exact space required for i keys and a hidden tag meaning $c_i = i$ for $i = 1, \dots, p-1$. The type p bucket will have the exact space for $m - 1$ keys, m pointers, and a hidden tag. Clearly, we have m types of buckets in this scheme ($p = m$) and $c_{p-1} = c_p$. The average of the linear combination $\mathcal{X}_n$ is given by

$$\mathbf{E}[\mathcal{X}_n] = \sum_{j=1}^{m-1} A_j [x^n](1-x)^{-\lambda_j} - \frac{a_p}{m-1} - \frac{(a_{p-1} - a_p)}{m + (-1)^m}(-1)^n \binom{m-1}{n}.$$

For large n

$$\mathbf{E}[\mathcal{X}_n] = A_1 n + \left(A_1 - \frac{a_p}{m-1}\right) + O(n^{\alpha-1}).$$

Hence

$$\begin{aligned}\mathbf{E}[X_n^{(i)}] &= \mathbf{E}[\mathcal{X}_n] \Big|_{\substack{a_i=1 \\ a_j=0;\ j\neq i}} \\ &\sim A_1^{(ii)} n.\end{aligned}$$

The factors $A_1^{(ii)}$ were obtained for $i = 1, \dots, m-2$ by Baeza-Yates [1] by a fringe analysis method (originated by Yao [25] in the context of B-trees and by Poblete and Munro [23] in the context of balancing heuristics for binary trees). For $i = 1, \dots, m-2$, Baeza-Yates found

$$A_1^{(ii)} = \frac{1}{(i+1)(i+2)(H_m - 1)}.$$

Thus we need only complete Baeza-Yates's calculations for the last two terms. We have

$$\begin{aligned}\sum_{j=1}^{m-1} \langle \lambda_j \rangle_k A_j^{(m-1,m-1)} &= k!\,\mathbf{E}[\mathcal{X}_k]\Big|_{\substack{a_{m-1}=1\\ a_j=0;\ j\neq m-1}} + \frac{(-1)^k}{m+(-1)^m}(m-1)_k \\ &= \frac{1}{m+(-1)^m}\langle -m+1\rangle_k, \qquad k=0,\dots,m-2,\end{aligned}$$

or, in matrix form,

$$\begin{aligned}&\begin{pmatrix} \langle\lambda_1\rangle_0 & \langle\lambda_2\rangle_0 & \cdots & \langle\lambda_{m-1}\rangle_0 \\ \langle\lambda_1\rangle_1 & \langle\lambda_2\rangle_1 & \cdots & \langle\lambda_{m-1}\rangle_1 \\ \vdots & \vdots & \ddots & \vdots \\ \langle\lambda_1\rangle_{m-2} & \langle\lambda_2\rangle_{m-2} & \cdots & \langle\lambda_{m-1}\rangle_{m-2} \end{pmatrix} \begin{pmatrix} A_1^{(m-1,m-1)} \\ A_2^{(m-1,m-1)} \\ \vdots \\ A_{m-1}^{(m-1,m-1)} \end{pmatrix} \\ &\quad = \frac{1}{m+(-1)^m} \begin{pmatrix} \langle -m+1\rangle_0 \\ \langle -m+1\rangle_1 \\ \vdots \\ \langle -m+1\rangle_{m-2} \end{pmatrix}.\end{aligned} \tag{6.1}$$

Similarly, $A_1^{(mm)}$ is obtained by solving the linear system

$$\begin{aligned}\sum_{j=1}^{m-1} \langle \lambda_j \rangle_k A_j^{(mm)} &= k!\,\mathbf{E}[\mathcal{X}_k]\Big|_{\substack{a_m=1\\ a_j=0;\ j\neq m}} + \frac{k!}{m-1} - \frac{(-1)^k}{m+(-1)^m}(m-1)_k \\ &= -\frac{1}{m+(-1)^m}\langle -m+1\rangle_k + \frac{k!}{m-1}, \qquad k=0,\dots,m-2,\end{aligned}$$

or, in matrix form,

$$\begin{aligned}&\begin{pmatrix} \langle\lambda_1\rangle_0 & \langle\lambda_2\rangle_0 & \cdots & \langle\lambda_{m-1}\rangle_0 \\ \langle\lambda_1\rangle_1 & \langle\lambda_2\rangle_1 & \cdots & \langle\lambda_{m-1}\rangle_1 \\ \vdots & \vdots & \ddots & \vdots \\ \langle\lambda_1\rangle_{m-2} & \langle\lambda_2\rangle_{m-2} & \cdots & \langle\lambda_{m-1}\rangle_{m-2} \end{pmatrix} \begin{pmatrix} A_1^{(mm)} \\ A_2^{(mm)} \\ \vdots \\ A_{m-1}^{(mm)} \end{pmatrix} \\ &\quad = -\frac{1}{m+(-1)^m} \begin{pmatrix} \langle -m+1\rangle_0 \\ \langle -m+1\rangle_1 \\ \vdots \\ \langle -m+1\rangle_{m-2} \end{pmatrix} + \frac{1}{m-1}\begin{pmatrix} \langle 1\rangle_0 \\ \langle 1\rangle_1 \\ \vdots \\ \langle 1\rangle_{m-2} \end{pmatrix}.\end{aligned} \tag{6.2}$$

Applying Cramer's rule to (6.1) yields

$$A_1^{(m-1,m-1)} = -\frac{1}{m+(-1)^m} \times \frac{\Delta_1}{\Delta},$$

where $\Delta = \det\left[\mathbf{V}_{m-1}(\lambda_1,\dots,\lambda_{m-1})\right]$ and Δ_1 is obtained from Δ by replacing its first column with the right-hand side vector in (6.1). After some elementary row operations, Δ_1 has the form of the Van Dermonde determinant $\det\left[\mathbf{V}_{m-1}(-m+1,\lambda_2,\dots,\lambda_{m-1})\right]$. Using the well-known expansion of the Van Dermonde determinant, one obtains

$$
\begin{aligned}
A_1^{(m-1,m-1)} &= \frac{\displaystyle\prod_{s=2}^{m-1}\big[\lambda_s-(-m+1)\big]}{\big[m+(-1)^m\big]\displaystyle\prod_{s=2}^{m-1}\big[(\lambda_s-\lambda_1)\big]}\\[2ex]
&= \frac{\displaystyle\prod_{s=2}^{m-1}\big[(-m+1)-\lambda_s\big]}{\big[m+(-1)^m\big]\displaystyle\prod_{s=2}^{m-1}\big[(2-\lambda_s)\big]}\\[2ex]
&= \frac{\psi(-m+1)}{[m+(-1)^m](-m-1)\psi'(2)}\\[2ex]
&= \frac{1}{m(m+1)(H_m-1)}.
\end{aligned}
$$

In a similar manner, one finds from (6.2)

$$
A_1^{(mm)} = \frac{\Delta_1}{\big[m+(-1)^m\big]\Delta} + \frac{\Delta_2}{(m-1)\Delta}
$$

where $\det \Delta_2 = \det\big[\mathbf{V}_{m-1}(1,\lambda_2,\dots,\lambda_{m-1})\big]$. After simplification, one obtains

$$
A_1^{(mm)} = \frac{1}{(m+1)(H_m-1)}.
$$

Assuming that we are dealing with an application where the keys are α bytes each on a computer whose memory addressing requires pointers of size β bytes each and the memory management system reserves ρ bytes for system usage, the actual average space S_n occupied by the random tree is

$$
\begin{aligned}
\mathbf{E}\big[S_n\big] &= \mathbf{E}\big[\big((m-1)\alpha+m\beta+\rho\big)X_n^{(m)}\big]+\sum_{i=1}^{m-1}\mathbf{E}\big[X_n^{(i)}(i\alpha+\rho)\big]\\
&\sim \big[(m-1)\alpha+m\beta+\rho\big]A_1^{(mm)}n+\sum_{i=1}^{m-1}(i\alpha+\rho)A_1^{(ii)}n\\
&= n\Bigg\{\big[(m-1)\alpha+m\beta+\rho\big]\frac{1}{(m+1)(H_m-1)} \qquad (6.3)\\
&\qquad+\alpha\sum_{i=1}^{m-1}\frac{i}{(i+1)(i+2)(H_m-1)}+\rho\sum_{i=1}^{m-1}\frac{1}{(i+1)(i+2)(H_m-1)}\Bigg\}\\
&= \left[\frac{\beta m}{m+1}+(H_m-1)\alpha+\frac{\rho}{2}\right]\frac{n}{H_m-1}.
\end{aligned}
$$

Under a fixed-bucket scheme for the same environment, each node is like a type m bucket of the exact fit scheme, i.e., the total actual space is asymptotic to

$$\frac{(m-1)\alpha + m\beta + \rho}{2(H_m - 1)} n.$$

Thus the asymptotic space of the exact fit scheme relative to the fixed-bucket scheme is

$$\frac{\text{average total space for exact fit}}{\text{average total space for fixed buckets}} = \frac{m\beta/(m+1) + \big(H_m - 1\big)\alpha + \rho/2}{\big[(m-1)\alpha + m\beta + \rho\big]/2}.$$

For large m, this is about

$$\frac{\beta + \rho/2 + \alpha \ln m}{(\alpha + \beta)m/2}.$$

As an example, in a personal computer environment, pointers are 4 bytes and each page of virtual memory under IBM's OS/2 operating system is 4K bytes. Under OS/2, $\rho = 1$ byte. Thus, a search tree implementation in this environment means a full node has 511 four-byte keys, a one-byte hidden tag, and 512 four-byte pointers. That is, $m = 512$, $\alpha = 4$, $\beta = 4$, and $\rho = 1$. In this instance, the relative efficiency between exact fit and fixed bucket is about $(9 + 36 \ln 2)/4096 \approx 8.4\%$. For small m ($m \le 26$), the reduction ratio is achieved with very high probability, in view of the small variance. In these cases, the asymptotic efficiency is obtained in probability since the space of each method is asymptotic to its average in probability.

(c) *Buddy System Scheme*. This memory management scheme allocates buckets with sizes that are proper powers of two. The smallest example that illustrates partial filling is the four-way branching with buckets of two, four, and eight words. We intend to present this instance in some detail to put all the procedures and distributions already discussed in perspective. This model may be suitable for internal memory use on a mainframe computer. In each bucket, one word of memory is reserved for system use. The largest bucket size must contain four pointers (four words on a mainframe computer). Thus the bucket with eight words can accommodate three keys. In terms of keys, the capacities of the three types of buckets involved are $c_1 = 1$, $c_2 = 3$, and $c_3 = 3$. The first two types are pointerless. The formula for the average of the linear combination $\mathcal{X}_n$ gives

$$\begin{aligned}\mathbf{E}\big[\mathcal{X}_n\big] &= \frac{A_1\langle\lambda_1\rangle_n}{n!} + \frac{A_2\langle\lambda_2\rangle_n}{n!} + \frac{A_3\langle\lambda_3\rangle_n}{n!} - \frac{1}{3}a_p - \frac{a_{p-1} - a_p}{5}(-1)^n\binom{3}{n} \\ &= A_1(n+1) + 2\Re\left\{\frac{A_2\langle\lambda_2\rangle_n}{n!}\right\} - \frac{1}{3}a_p - \frac{a_{p-1} - a_p}{5}(-1)^n\binom{3}{n}.\end{aligned}$$

The function $h_1(x)$ that appears in the linear system for the A's is given by

$$h_1(x) = -\frac{a_p}{3(1-x)} - \frac{(a_{p-1} - a_p)}{5}(1-x)^3.$$

The characteristic equation is $\lambda(\lambda+1)(\lambda+2) - 24 = 0$ and has the roots

$$\lambda_1 = 2, \qquad \lambda_2 = \frac{-5 + \sqrt{23}\,i}{2}, \qquad \lambda_3 = \frac{-5 - \sqrt{23}\,i}{2}.$$

Here, $G_1(0) = 0$, $G_1'(0) = \mathbf{E}\big[\mathcal{X}_1\big] = a_1$, $G_1''(0) = 2\mathbf{E}\big[\mathcal{X}_2\big] = 2a_2$. Thus the constants of integration A_1, A_2, and A_3 are determined by solving the linear system

$$\begin{pmatrix} 1 & 1 & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1(\lambda_1+1) & \lambda_2(\lambda_2+1) & \lambda_3(\lambda_3+1) \end{pmatrix} \begin{pmatrix} A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

$$= \begin{pmatrix} 0 \\ a_1 \\ 2a_2 \end{pmatrix} - \begin{pmatrix} -\frac{1}{5}a_2 - \frac{2}{15}a_3 \\ \frac{3}{5}a_2 - \frac{14}{15}a_3 \\ -\frac{6}{5}a_2 + \frac{8}{15}a_3 \end{pmatrix}. \tag{6.4}$$

Solving this linear system with $a_1 = 1$, $a_2 = a_3 = 0$, one gets

$$A_1^{(11)} = \frac{2}{13}, \qquad A_2^{(11)} = -\frac{1}{13} - \frac{4\sqrt{23}}{299}i, \qquad A_3^{(11)} = -\frac{1}{13} + \frac{4\sqrt{23}}{299}i;$$

thus,

$$\mathbf{E}\big[X_n^{(1)}\big] = \frac{2}{13}(n+1) + \frac{2}{n!}\Re\left\{\left(-\frac{1}{13} - \frac{4\sqrt{23}}{299}i\right)\left\langle\frac{-5+\sqrt{23}\,i}{2}\right\rangle_n\right\} \sim \frac{2}{13}n.$$

Similar calculations with $a_1 = 0$, $a_2 = 1$, $a_3 = 0$, and $a_1 = a_2 = 0$, $a_3 = 1$, lead to

$$\mathbf{E}\big[X_n^{(2)}\big] \sim \frac{8}{65}n \quad \text{and} \quad \mathbf{E}\big[X_n^{(3)}\big] \sim \frac{12}{65}n,$$

respectively. Consequently, the asymptotic average number of conversions of different types are

$$\mathbf{E}\big[C_n^{(1)}\big] \sim \frac{6}{13}n, \qquad \mathbf{E}\big[C_n^{(2)}\big] \sim \frac{4}{13}n, \qquad \mathbf{E}\big[C_n^{(3)}\big] \sim \frac{12}{65}n.$$

The formula for the variances of the linear combination $\mathcal{X}_n$ follows from

$$\mathbf{Var}\big[\mathcal{X}_n\big] = \mathbf{E}\big[\mathcal{X}_n^2\big] - \mathbf{E}^2\big[\mathcal{X}_n\big],$$

where $\mathbf{E}\big[\mathcal{X}_n\big]$ has already been determined and $\mathbf{E}\big[\mathcal{X}_n^2\big]$ is given by

$$\begin{aligned}
\mathbf{E}\big[\mathcal{X}_n^2\big] = {} & K_1(n+1) + 2\Re\left\{\frac{K_2\langle\lambda_2\rangle_n}{n!}\right\} + \frac{a_p^2}{9} \\
& + (-1)^{n+1}\frac{(a_{p-1}^2 - a_p^2)}{5}\binom{3}{n} + \frac{4}{325}(-1)^{n+1}(a_{p-1} - a_p)^2\binom{7}{n} \\
& + \frac{6(a_{p-1} - a_p)A_1}{5} \times \frac{\langle -2\rangle_n}{n!} \\
& - \frac{288(a_{p-1} - a_p)}{5}\Re\left\{\frac{A_2}{\psi(\lambda_2 - 4)} \times \frac{\langle\lambda_2 - 4\rangle_n}{n!}\right\} \\
& + A_1^2 n^2 + 3A_1^2 n + 2A_1^2 + 4A_1\Re\left\{(\lambda_2 + n)A_2\frac{\langle\lambda_2\rangle_n}{n!}\right\} \\
& + 144\Re\left\{\frac{A_2^2}{\psi(\lambda_{22})} \times \frac{\langle\lambda_{22}\rangle_n}{n!}\right\} + \frac{144|A_2|^2}{\psi(2\Re\lambda_2 - 1)} \times \frac{\langle 2\Re\lambda_2 - 1\rangle_n}{n!}.
\end{aligned}$$

The function $h_2(x)$ that appears in the linear systems for second moments is

$$\begin{aligned} h_2(x) &= \frac{a_p^2}{9(1-x)} - \frac{a_{p-1}^2 - a_p^2}{5}(1-x)^3 - \frac{4}{325}(a_{p-1}-a_p)^2(1-x)^7 \\ &+ \frac{6(a_{p-1}-a_p)A_1}{5}(1-x)^2 - \sum_{j=2}^{3} \frac{144(a_{p-1}-a_p)A_j}{5\psi(\lambda_j-4)(1-x)^{\lambda_j-4}} + \frac{2A_1^2}{(1-x)^3} \\ &+ \frac{2\lambda_2 A_1 A_2}{(1-x)^{\lambda_2+1}} + \frac{2\lambda_3 A_1 A_3}{(1-x)^{\lambda_3+1}} + \sum_{2\le j,k\le 3} \frac{72 A_j A_k}{(1-x)^{\lambda_{jk}}\psi(\lambda_{jk})}. \end{aligned}$$

Here, $G_2(0) = 0$, $G_2'(0) = \mathbf{E}[\mathcal{X}_n^2] = a_1^2$, $G_2''(0) = 2\mathbf{E}[\mathcal{X}_n^2] = 2a_2^2$. Thus the constants of integration K_1, K_2, and K_3 are determined by solving the linear system

$$\begin{pmatrix} 1 & 1 & 1 \\ \lambda_1 & \lambda_2 & \lambda_3 \\ \lambda_1(\lambda_1+1) & \lambda_2(\lambda_2+1) & \lambda_3(\lambda_3+1) \end{pmatrix} \begin{pmatrix} K_1 \\ K_2 \\ K_3 \end{pmatrix} = \begin{pmatrix} 0 \\ a_1^2 \\ 2a_2^2 \end{pmatrix} - \begin{pmatrix} h_2(0) \\ h_2'(0) \\ h_2''(0) \end{pmatrix}. \tag{6.5}$$

Solving this linear system with $a_1 = 1$, $a_2 = a_3 = 0$, one gets

$$K_1^{(11)} = \frac{31}{377}, \qquad K_2^{(11)} = \frac{-35-\sqrt{23}\,i}{208}, \qquad K_3^{(11)} = \frac{-35+\sqrt{23}\,i}{208};$$

thus,

$$\begin{aligned} \mathbf{Var}\big[X_n^{(1)}\big] &= \frac{4}{169}n^2 + \frac{751}{4901}n + \frac{635}{4901} + (-1)^{n+1}\frac{3}{299}\binom{6}{n} \\ &+ \frac{2}{n!}\Re\left\{\left(\frac{-35-\sqrt{23}\,i}{208}\right)\left\langle\frac{-5+\sqrt{23}\,i}{2}\right\rangle_n\right\} \\ &+ \frac{8}{13(n!)}\Re\left\{\left(-\frac{1}{13} - \frac{4\sqrt{23}}{299}i\right)\left(\frac{-5+\sqrt{23}\,i}{2} + n\right)\right. \\ &\qquad\qquad \left.\left\langle\frac{-5+\sqrt{23}\,i}{2}\right\rangle_n\right\} \\ &+ \frac{144}{n!}\Re\left\{\left(\frac{1199+139\sqrt{23}\,i}{43285632}\right)\left\langle -6+\sqrt{23}\,i\right\rangle_n\right\} \\ &- \left[\frac{2}{13}(n+1) + \frac{2}{n!}\Re\left\{\left(-\frac{1}{13} - \frac{4\sqrt{23}}{299}i\right)\left\langle\frac{-5+\sqrt{23}\,i}{2}\right\rangle_n\right\}\right]^2 \\ &\sim \frac{519}{4901}n. \end{aligned}$$

Similar calculations with various settings for a_1, a_2, and a_3 lead to

$$\mathbf{Var}\big[X_n^{(2)}\big] \sim \frac{7586}{122525}n, \qquad \mathbf{Var}\big[X_n^{(3)}\big] \sim \frac{2276}{122525}n,$$

$$\mathbf{Var}\big[X_n^{(1)} + X_n^{(2)}\big] \sim \frac{5276}{122525}n,$$

$$\mathbf{Var}\big[X_n^{(2)} + X_n^{(3)}\big] \sim \frac{126}{4901}n,$$

$$\mathbf{Var}\big[X_n^{(3)} + X_n^{(1)}\big] \sim \frac{19061}{122525}n.$$

Hence, the asymptotic variance-covariance matrix for $\mathbf{X}_n$ is

$$\mathbf{\Lambda}_3 n = \begin{pmatrix} \frac{519}{4901} & -\frac{3057}{49010} & \frac{381}{24505} \\ -\frac{3057}{49010} & \frac{7586}{122525} & -\frac{3356}{122525} \\ \frac{381}{24505} & -\frac{3356}{122525} & \frac{2276}{122525} \end{pmatrix} n$$

and the corresponding asymptotic variance-covariance matrix for $\mathbf{C}_n$ is

$$\mathbf{J}_3\mathbf{\Lambda}_3\mathbf{J}_3^T n = \begin{pmatrix} \frac{186}{4901} & -\frac{207}{9802} & \frac{33}{4901} \\ -\frac{207}{9802} & \frac{126}{4901} & -\frac{216}{24505} \\ \frac{33}{4901} & -\frac{216}{24505} & \frac{2276}{122525} \end{pmatrix} n.$$

An exact computation of all variances and covariances for this example was carried out; for details of these computations, see [14]. The essential behavior in all averages, variances and covariances, is a linear component in n accompanied by very small periodic oscillations. In view of Theorem 2, the vector $(\hat{X}_n^{(1)}, \hat{X}_n^{(2)}, \hat{X}_n^{(3)})^T$ converges in distribution to a multinormal distribution with variance-covariance matrix $\mathbf{\Lambda}_3$ and the vector $(\hat{C}_n^{(1)}, \hat{C}_n^{(2)}, \hat{C}_n^{(3)})^T$ converges in distribution to a multinormal distribution with variance-covariance matrix $\mathbf{J}_3\mathbf{\Lambda}_3\mathbf{J}_3^T$. More generally, for higher branch factor m, the buckets are of size 2^k words, with one reserved in each bucket for system use. With numeric keys occupying one word each and one-word long pointers, the pointerless buckets have capacities $c_i = 2^i - 1$, $i = 1, \ldots, p-1$, and contain a hidden tag. Like type $p-1$, the last bucket type, type p, contains a hidden tag and has capacity $c_p = 2^{p-1} - 1$, but it also contains 2^{p-1} pointers. By lumping together nodes in the exact fit scheme that are collapsed into one bucket under the buddy system, we can obtain the relative space economy achieved by the buddy system allocation scheme. Nodes containing $2^{k-1}, 2^{k-1}+1, \ldots, 2^k - 1$ in the exact fit scheme are all implemented as buckets of type k with capacity $c_k = 2^k - 1$ under the buddy system. The space is not tight like the perfect packing of the exact fit scheme, but for small k (these are the majority of nodes, on average) only a small waste is still an improvement over the severe waste of the fixed-bucket scheme. If $\tilde{X}_n^{(j)}$ is the number of nodes holding exactly j keys under exact fit, then the average number of nodes under the buddy system, for type k, $k = 1, 2, \ldots, p-1$, is

$$\begin{aligned} \mathbf{E}\big[X_n^{(k)}\big] &= \sum_{j=2^{k-1}}^{2^k-1} \mathbf{E}\big[\tilde{X}_n^{(j)}\big], \qquad k = 1, 2, \ldots, p-1, \\ &\sim \sum_{j=2^{k-1}}^{2^k-1} \frac{n}{(j+1)(j+2)(H_m - 1)} \\ &\sim \Big(\frac{n}{H_{m-1} - 1}\Big)\Big(\frac{1}{2^{k-1}+1} - \frac{1}{2^k+1}\Big) \end{aligned}$$

and the average number of nodes of type p is

$$\mathbf{E}[X_n^{(p)}] \sim \frac{n}{(m+1)(H_m-1)}.$$

Thus, S_n, the total actual space, has an average

$$\begin{aligned}
\mathbf{E}[S_n] &= \sum_{k=1}^{p} 2^k \mathbf{E}[X_n^{(k)}] \\
&\sim \frac{n}{(H_m-1)}\left[\sum_{k=1}^{p-1} 2^k\left(\frac{1}{2^{k-1}+1} - \frac{1}{2^k+1}\right) + \frac{2^p}{m+1}\right] \\
&= \frac{n}{(H_{2^{p-1}}-1)}\left(p-1+\frac{2^{p-1}}{2^{p-1}+1} - \sum_{k=1}^{p-2}\frac{1}{2^k+1}\right) \\
&\approx \frac{n}{(H_{2^{p-1}}-1)}\left(p-0.7645\right), \quad \text{for large } p \\
&\approx \frac{n}{\ln 2} \\
&\approx 1.44n.
\end{aligned}$$

We compare this scheme to the previous two schemes under the same environment (i.e., same assumptions concerning the space for data, pointers, and hidden tag) and same branch factor. Compared with the fixed-bucket scheme, for which there is an average $\frac{n}{2}(H_{2^{p-1}}-1)^{-1}$ nodes, each requiring 2^p bytes, the efficiency of the buddy system is $p/2^{p-1}$, which tends to 0 fast as $p \to \infty$. Compared to the exact fit scheme, when we take $\alpha = 1$ word in formula (6.3) (i.e., when (6.3) is interpreted in units of words instead of bytes), the efficiency for large m is about

$$\begin{aligned}
\frac{\left(p/(H_m-1)\right)n}{\left(\alpha H_m/(H_m-1)\right)n} &\approx \frac{p}{\ln m} = \frac{p}{\ln 2^{p-1}} = \frac{p}{(p-1)\ln 2} \\
&\approx \frac{1}{\ln 2} \\
&\approx 1.44.
\end{aligned}$$

This method is only 44% worse than exact fit on average but may suit computer systems where exact fit is not available. The filling ratio $(\ln 2)^{-1} = 1.44$ has been observed in a number of data structures in comparison with the perfect packing and appears to be a universal constant underlying data retrieval methods. Instances may be found in digital search trees (see [7]), quadtrees (see [10]), B-trees (see [25] and [5]), hashing schemes (see [13] and [6]), and tries (see [11] and [24]). These results are surveyed in [9].

7. Conclusion. A random search tree with branch factor m and elastic buckets for keys was discussed, and the case of buckets with general capacities was examined. Formulas were derived for the expectations and variances of $X_n^{(i)}$, the number of buckets of type i, and for the covariances between different types. For $3 \le m \le 26$, all variances and covariances were found to be asymptotically linear in n, but for higher branch factors they experience a phase transition and become a superlinear (but subquadratic) function of n. This phase transition persists in all practical memory management schemes and seems to be independent of the bucket sizes. The joint distribution of $(\hat{X}_n^{(1)}, \dots, \hat{X}_n^{(p)})^T$, a vector of the normalized random variables for $X_n^{(i)}$, was found to be asymptotically jointly multinormal for $3 \le m \le 26$.

An illustrating example of the buddy system scheme, the four-way branching case with bucket sizes of two, four, and eight words for data, pointers, and a hidden tag for system use, is worked out in detail. For large branch factor, this method is shown to be only 44% worse than exact fit but may suit computer systems where exact fit is not available.

An associated problem, the study of the number of conversions between bucket types can also be found from the behavior of the random variables $\mathcal{X}_n^{(i)}$. In practical memory management schemes, knowledge of the number of conversions between bucket types may be of use in determining the associated cost in the storage and handling of data. The same properties exhibited by $\mathbf{X}_n$, a phase transition of all variances and covariances for $m > 26$ and asymptotic multinormality in the range $3 \le m \le 26$, are also inherent in the conversions problem.

Appendix. The purpose of this appendix is to compute the particular solution, $h_2(x)$, of the equation

$$\frac{d^{m-1}}{dx^{m-1}}G_2(x) = \frac{m!\,G_2(x)}{(1-x)^{m-1}} + H_2(x),$$

where

$$H_2(x) \overset{\text{def}}{=} \frac{m!\,(m-1)G_1^2(x)}{(1-x)^{m-2}} + \frac{(m-1)!\,a_p^2}{(1-x)^m} + \frac{2a_p(m!)G_1(x)}{(1-x)^{m-1}}$$
$$+ (m-1)!\,(a_{p-1}^2 - a_p^2)\delta_{c_p,c_{p-1}}.$$

Observe that all the terms of $H_2(x)$ have the general form $c(1-x)^{-\lambda-m+1}$. Each term of $H_2(x)$ of the form $c(1-x)^{-\lambda-m+1}$ contributes to the particular solution $h_2(x)$ a term of the form

$$\frac{c}{\psi(\lambda)}(1-x)^{-\lambda},$$

provided that $\psi(\lambda) \ne 0$. If $\psi(\lambda) = 0$, then a term of the form

$$\frac{c}{\psi'(\lambda)}(1-x)^{-\lambda}\ln\left(\frac{1}{1-x}\right)$$

is contributed to the particular solution. Thus, $h_2(x)$ and its contribution to $\mathbf{E}[\mathcal{X}_n^2]$ can be obtained by organizing the terms of $H_2(x)$ into the following cases:

(i) $(m-1)!\,a_p^2(1-x)^{-m}$. For this term $\lambda = 1$ with $\psi(1) = -(m-1)!\,(m-1) \ne 0$ for all $m \ge 2$. Thus this term contributes

$$-\frac{a_p^2}{(m-1)(1-x)}$$

to the particular solution and adds

$$-\frac{a_p^2}{m-1}$$

to $\mathbf{E}[\mathcal{X}_n^2]$.

(ii) $(m-1)!\,(a_{p-1}^2 - a_p{}^2)\delta_{c_p,c_{p-1}}$. For this term

$$\lambda = -m+1, \qquad \psi(\lambda) = -(m-1)!\,\big[m+(-1)^m\big] \neq 0,$$

for all $m \geq 2$. Thus this term contributes

$$-\frac{(a_{p-1}^2 - a_p{}^2)\delta_{c_p,c_{p-1}}}{\big[m+(-1)^m\big]}(1-x)^{m-1}$$

to the particular solution which adds

$$\frac{(-1)^{n+1}(a_{p-1}^2 - a_p{}^2)\delta_{c_p,c_{p-1}}}{\big[m+(-1)^m\big]}\binom{m-1}{n}$$

to $\mathbf{E}\big[\mathcal{X}_n^2\big]$, a term that disappears for $n \geq m$.

(iii) $2a_p(m!)G_1(x)(1-x)^{-m+1}$. Using the solution for $G_1(x)$ from §3, this term presents three subcases as follows:

(a) $2a_p(m!)(1-x)^{-m+1}\big[-a_p(m-1)^{-1}(1-x)^{-1}\big] = -2a_p^2(m!)(m–1)^{-1}(1-x)^{-m}$. For this term $\lambda = 1$ with $\psi(1) = -(m-1)!\,(m-1) \neq 0$, for all $m \geq 2$. Thus this term contributes

$$\frac{2a_p^2 m}{(m-1)^2(1-x)}$$

to the particular solution which adds

$$\frac{2a_p^2 m}{(m-1)^2}$$

to $\mathbf{E}\big[\mathcal{X}_n^2\big]$.

(b) $-2a_p(m!)\delta_{c_p,c_{p-1}}(a_{p-1} - a_p)\big[m+(-1)^m\big]^{-1}$. This term cancels out with a similar term that appears in $m!\,(m-1)G_1^2(x)(1-x)^{-m+2}$.

(c) $2a_p(m!)(1-x)^{-m+1}\big[A_j(1-x)^{-\lambda_j}\big]$. These terms cancel out with similar terms that appear in $m!\,(m-1)G_1^2(x)(1-x)^{-m+2}$.

(iv) $m!\,(m-1)G_1^2(x)(1-x)^{-m+2}$. Using the solution for $G_1(x)$ from §3, this term presents six subcases as follows:

(a) $m!\,a_p^2(m-1)^{-1}(1-x)^{-m}$. For this term $\lambda = 1$ with $\psi(1) = -(m-1)!\,(m-1) \neq 0$ for all $m \geq 2$. Thus this term contributes

$$-\frac{ma_p^2}{(m-1)^2(1-x)}$$

to the particular solution, which adds

$$-\frac{ma_p^2}{(m-1)^2}$$

to $\mathbf{E}\big[\mathcal{X}_n^2\big]$.

(b) $m!\,(m-1)\delta^2_{c_p,c_{p-1}}(a_{p-1}-a_p)^2\big[m+(-1)^m\big]^{-2}(1-x)^m$. For this term $\lambda = -2m+1$ but the smallest real part of any root of $\psi(\lambda)$ is no less than $-m$, so $-2m+1$ cannot be a root. Thus this term contributes

$$\frac{m!\,(m-1)\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)^2(1-x)^{2m-1}}{\big[m+(-1)^m\big]^2\psi(-2m+1)}$$

to the particular solution, which adds

$$(-1)^n\frac{m!\,(m-1)\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)^2}{\big[m+(-1)^m\big]^2\psi(-2m+1)}\binom{2m-1}{n}$$

to $\mathbf{E}\big[\mathcal{X}_n^2\big]$, a term that disappears for $n \geq 2m$.

(c) $-2a_p(m!)(1-x)^{-m+2}\big[A_j(1-x)^{-\lambda_j-1}\big]$. These terms cancel out with the terms in subcase iii(c).

(d) $2a_p(m!)\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)\big[m+(-1)^m\big]^{-1}$. This term cancels out with the term in subcase iii(b).

(e) $m!\,(m-1)\big[-2\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)A_j\big\{m+(-1)^m\big\}^{-1}(1-x)^{-\lambda_j+1}\big]$. For these terms $\lambda = \lambda_j - m$ and the contribution, $\mu_j(x)$, to the particular solution is of the form

$$\mu_j(x) = \begin{cases} \theta_j(1-x)^{-\lambda_j+m}\big\{\psi(\lambda_j-m)\big\}^{-1}, & \psi(\lambda_j-m)\neq 0,\\ \theta_j(1-x)^{-\lambda_j+m}\big\{\psi'(\lambda_j-m)\big\}^{-1}\ln\Big(\dfrac{1}{1-x}\Big), & \psi(\lambda_j-m)=0. \end{cases}$$

where $\theta_j = -2(m!)(m-1)\big[\delta_{c_p,c_{p-1}}(a_{p-1}-a_p)A_j\big\{m+(-1)^m\big\}^{-1}\big]$. The added contribution is $O\big(n^{\Re\lambda_j-m-1+\epsilon}\big)$ for every $\epsilon > 0$. But $\Re\lambda_j \leq 2$ for $j = 1, \dots, m-1$, i.e., this term is only $O\big(n^\epsilon/n^{m-1}\big)$. As $m \geq 2$, this contribution diminishes if $\epsilon > 0$ is small enough; for $\epsilon < m-2$ this term is $O\big(n^{-1}\big)$ as $n \to \infty$.

(f) $m!\,(m-1)A_jA_k(1-x)^{-\lambda_j-\lambda_k-m+2}$. Setting $\lambda_{jk} = \lambda_j + \lambda_k - 1$ puts these terms into the form $m!\,(m-1)A_jA_k(1-x)^{-\lambda_{jk}-m+1}$. For these terms $\lambda = \lambda_{jk}$ and the contribution, $\eta_{jk}(x)$, to the particular solution is of the form

$$\eta_{jk}(x) = \begin{cases} \beta_{jk}(1-x)^{-\lambda_{jk}}\big\{\psi(\lambda_{jk})\big\}^{-1}, & \psi(\lambda_{jk})\neq 0,\\ \beta_{jk}(1-x)^{-\lambda_{jk}}\big\{\psi'(\lambda_{jk})\big\}^{-1}\ln\Big(\dfrac{1}{1-x}\Big), & \psi(\lambda_{jk})=0, \end{cases}$$

where $\beta_{jk} = m!\,(m-1)A_jA_k$. The contribution of these terms into $\mathbf{E}\big[\mathcal{X}_n^2\big]$ can be classified into four categories:

(1) The case $j = 1, k = 1$. For this pair of indices $\lambda_{11} = 3$, $\psi(3) = m!\,(m-1)/2$, and the contribution, $\eta_{11}(x)$, to the particular solution is

$$\eta_{11}(x) = \frac{2A_1^2}{(1-x)^3},$$

which adds

$$A_1^2(n+1)(n+2) = A_1^2n^2 + 3A_1^2n + 2A_1^2$$

to $\mathbf{E}\big[\mathcal{X}_n^2\big]$.

(2) The case $j = 1, k > 1$ (symmetric with this case is $k = 1$ and $j > 1$). For this pair of indices

$$\lambda_{1k} = \lambda_k + 1, \qquad \psi(\lambda_k + 1) = m!\,(m-1)\lambda_k^{-1} \neq 0,$$

for any $m \geq 2$ and the contribution, $\eta_{1k}(x)$, to the particular solution is

$$\frac{\lambda_k A_1 A_k}{(1-x)^{\lambda_k+1}},$$

which adds

$$A_1 A_k (\lambda_k + n) \frac{\langle \lambda_k \rangle_n}{n!}$$

to $\mathbf{E}[\mathcal{X}_n^2]$. The collective contribution of all such terms to the expectation is

$$2A_1 \sum_{k=2}^{m-1} A_k (\lambda_k + n) \frac{\langle \lambda_k \rangle_n}{n!} = 2A_1 \sum_{k=2}^{m-1} \lambda_k \frac{A_k \langle \lambda_k \rangle_n}{n!} + 2A_1 n \sum_{k=2}^{m-1} \frac{A_k \langle \lambda_k \rangle_n}{n!}.$$

The first term on the right is $O(n^{\alpha-1})$, the second appears in $\mathbf{E}^2[\mathcal{X}_n]$, and thus this order of magnitude disappears in $\mathbf{Var}[\mathcal{X}_n] = \mathbf{E}[\mathcal{X}_n^2] - \mathbf{E}^2[\mathcal{X}_n]$.

(3) The case $2 \leq j, k \leq 3$. These terms are the causes of the aforementioned phase transitions in the variances and covariances since its behavior for $m \leq 26$ is different from that for $m > 26$ as discussed next. For each of these terms and whether $\ln\left(\frac{1}{1-x}\right)$ appears in it or not, if $\alpha < 3/2$, then $\Re\lambda_{jk} = 2\alpha - 1 < 2$ and for every $\epsilon > 0$.

$$\begin{aligned}[x^n]\eta_{jk}(x) &= O\left(n^{\Re\lambda_{jk}-1+\epsilon}\right) \\ &= o(n),\end{aligned}$$

where the last equality holds for small enough ϵ. But if $\alpha > 3/2$, we have $\Re\lambda_{jk} = 2\alpha - 1 > 2$ and so $\psi(\lambda_{jk}) \neq 0$ and the contributed solution is

$$m!\,(m-1) \sum_{2 \leq j,k \leq 3} \frac{A_j A_k}{\psi(\lambda_{jk})} (1-x)^{-\lambda_{jk}}.$$

By the Stirling approximation of the Gamma function, this adds to $\mathbf{E}[\mathcal{X}_n^2]$ the term

$$c_m(n) \stackrel{\text{def}}{=} n^{2\alpha-2} \sum_{2 \leq j,k \leq 3} A_j A_k \frac{m!\,(m-1)}{\psi(\lambda_{jk})\Gamma(\lambda_{jk})} n^{i\Im\lambda_{jk}},$$

where $i = \sqrt{-1}$. This growth is superlinear for $\alpha > 3/2$ (but subquadratic as $\alpha < 2$ for all m). The function $c_m(n)$ is $\frac{\pi}{\beta}$-periodic in $\ln n$.

(4) All the other indices: For these $\lambda_{jk} = \lambda_j + \lambda_k - 1$ and at least one index, j or k, has to be greater than 2, i.e., $\Re\lambda_{jk} < 2\alpha - 1$. The corresponding solution therefore (whether it has a logarithmic term or not) contributes $o\left(n^{2\alpha-2+\epsilon}\right)$, for every $\epsilon > 0$. Putting all of the above contributions to $\mathbf{E}[\mathcal{X}_n^2]$ together and then subtracting $\mathbf{E}^2[\mathcal{X}_n]$, we get the variance as in Theorem 1.

Acknowledgment. Our original proof of the asymptotic normality covered the range of $3 \leq m \leq 15$. The authors are indebted to Dr. Robert T. Smythe for suggesting an extension of the proof for Theorem 2 to work in the entire range of the linearity of variances and covariances ($3 \leq m \leq 26$). We also wish to thank Ricardo Baeza-Yates for updating some references and for several other helpful comments.

REFERENCES

[1] R. BAEZA-YATES, *Some average measures in m-ary search trees*, Inform. Process. Lett., 25 (1987), pp. 375–381.

[2] R. BAEZA-YATES AND P. LARSON, *Performance of B^+-trees with partial expansions*, IEEE Trans. Knowledge Data Engrg., 1 (1989), pp. 248–257.

[3] P. BILLINGSLEY, *Convergence of Probability Measures*, John Wiley, New York, (1968).

[4] L. DEVROYE, *On the height of random m-ary search trees*, Random Structures Algorithms, 1 (1990), pp. 191–203.

[5] B. EISENBARTH, N. ZIVIANI, G. GONNET, K. MEHLHORN, AND D. WOOD, *The theory of fringe analysis and its application to 2–3 trees and B-trees*, Inform. Control, 55 (1982), pp. 125–174.

[6] R. FAGIN, J. NIEVERGELT, N. PIPPENGER, AND H. SRONG, *Extendible hashing—a fast access method for dynamic files*, ACM Trans. Database Systems, 4 (1979), pp. 315–344.

[7] P. FLAJOLET AND B. RICHMOND, *Generalized digital trees and their difference-differential equations*, Random Structures Algorithms, 3 (1992), pp. 305–320.

[8] G. FREDERICKSON, *Improving storage utilization in balanced trees*, Proc. Seventeenth Allerton Conf. Commun. Contr. Comput., Monticello, IL, 1979, pp. 255–264.

[9] G. GONNET AND R. BAEZA-YATES, *Handbook of Algorithms and Data Structures*, 2nd edition, Addison-Wesley, Reading, MA, 1991.

[10] M. HOSHI AND P. FLAJOLET, *Page usage in quadtree indexes*, BIT, 32 (1992), pp. 384–402.

[11] P. JACQUET AND M. RÉGNIER, *Normal limiting distribution of the size of tries*, Proceedings of the Twelvth IFIP WG 7.3 Symposium on Computer Performance Modelling, Measurement, and Evaluation, December 7-9, 1987, Brussels, Belgium, P. J. Courtois and G. Latouche, eds., North Holland, Amsterdam, 1988.

[12] D. KNUTH, *The Art of Computer Programming: Sorting and Searching*, Vol. 3. Addison-Wesley, Reading, MA, 1973.

[13] P. LARSON, *Dynamic hashing*, BIT, 18 (1978), pp. 184–201.

[14] W. LEW, AND H. MAHMOUD, *The joint distribution of elastic buckets in a data structure*, Tech. Report #93-5, Department of Statistics/Computer & Information Systems, The George Washington University, Washington, D.C., 1993.

[15] D. LOMET, *Partial expansions for file organizations with an index*, ACM Trans. Database Systems, 12 (1987), pp. 65–84.

[16] H. MAHMOUD, *On the average internal path length of m-ary search trees*, Acta Inform., 23 (1986), pp. 111–117.

[17] ———, *Evolution of Random Search Trees*, John Wiley, New York, 1992.

[18] ———, *A law of large numbers for path lengths in search trees*, Random Graphs, Vol. 2, John Wiley, New York, 1992.

[19] H. MAHMOUD AND B. PITTEL, *On the most probable shape of a search tree grown from a random permutation*, SIAM J. Algebraic Discrete Methods, 5 (1984), pp. 69–81.

[20] ———, *On the joint distribution of the insertion path length and the number of comparisons in search trees*, Discrete Appl. Math., 20 (1988), pp. 243–251.

[21] ———, *Analysis of the space of search trees under the random insertion algorithm*, J. Algorithms, 10 (1989), pp. 52–75.

[22] B. PITTEL, *Note on the heights of random recursive trees and random m-ary search trees*, Random Structures and Algorithms, 5 (1994), pp. 337–347.

[23] P. POBLETE AND I. MUNRO, *The analysis of a fringe heuristic for binary search trees*, J. Algorithms, 6 (1985), pp. 336–350.

[24] M. RÉGNIER, AND P. JACQUET, *New results on the size of tries*, IEEE Trans. Inform. Theory, IT-35 (1989), pp. 203–205.

[25] A. YAO, *On random 2–3 trees*, Acta Inform., 9 (1978), pp. 159–170.

SIAM J. COMPUT.
Vol. 23, No. 5, pp. 1075–1091, October 1994

TIGHT BOUNDS ON THE COMPLEXITY OF THE BOYER-MOORE STRING MATCHING ALGORITHM*

RICHARD COLE†

Abstract. The problem of finding all occurrences of a pattern of length m in a text of length n is considered. It is shown that the Boyer–Moore string matching algorithm performs roughly $3n$ comparisons and that this bound is tight up to $O(n/m)$; more precisely, an upper bound of $3n - 3(n-m+1)/(m+2)$ comparisons is shown, as is a lower bound of $3n(1-o(1))$ comparisons, as $\frac{n}{m} \to \infty$ and $m \to \infty$. While the upper bound is somewhat involved, its main elements provide a simple proof of a $4n$ upper bound for the same algorithm.

Key words. string matching, character comparisons, Boyer–Moore algorithm, amortized analysis

AMS subject classifications. 68Q20, 68Q25, 68R15

1. Introduction.

String matching is the problem of finding a pattern of length m in a text of length n; often all occurrences of the pattern are sought. This problem is well studied and is a staple of textbooks on algorithms (for instance, [3], [5], [12]). It is an important subproblem in a number of domains including text editing, symbol manipulation, and data retrieval.

The best known algorithms for this problem are the Knuth–Morris–Pratt algorithm [19] and the Boyer-Moore algorithm [7] (we refer to these as the KMP and BM algorithms, respectively). Both these algorithms are linear time; the bound for the KMP algorithm is very straightforward, the bound for the BM algorithm is considerably less so. An interesting aspect of the BM algorithm is that on average (in probabilistic settings) it takes sublinear time; this effect is observed in practice too. A recent study of this behavior is given in [6].

Many other types of string matching algorithms have been studied; these include matching several strings simultaneously [1], real-time matching [15], matching in constant space [17], [13], randomized algorithms [20], parallel algorithms [26], [27], approximate matching [21], [16], [25], and two-dimensional matching [4].

Both the KMP and BM algorithms begin by computing a shift function. (We review the shift function for the BM algorithm later.) Following the precomputation of the shift function, the actual match is carried out. The complexity of the algorithm is usually stated in terms of the number of comparisons required for the matching stage (excluding the precomputation stage). The KMP algorithm requires $2n - m$ comparisons in the worst case and this is a tight bound for $m \geq 2$. For the BM algorithm, the first linear bound was given by Knuth in [19] as a bound of $7n$ comparisons (often misquoted as $6n$ comparisons); this proof is difficult. In 1980, Guibas and Odlyzko gave another proof [18], obtaining a bound of $4n$ comparisons; their proof is also nontrivial. Guibas and Odlyzko also conjectured that $2n$ comparisons might be the correct bound. Our contribution is twofold.

- We give a bound, tight up to lower-order terms, of roughly $3n$ comparisons, thereby disproving the just-mentioned conjecture.
- In addition, the basic elements of this proof provide a direct and straightforward demonstration of an upper bound of $4n$ comparisons.

The above bounds for the BM algorithm assume that the pattern is not *periodic*, i.e., of the form wv^k, where w is a proper suffix of v and $k \geq 2$. Galil [14] showed how to modify the BM algorithm so that a linear bound applies in this case too; in fact, using essentially Galil's modification, our bounds apply unchanged to such patterns.

*Received by the editors February 20, 1991; accepted for publication (in revised form) May 5, 1993. Supported in part by National Science Foundation grants CCR-8902221, CCR-8906949, and CCR-9200290. A preliminary version of this paper appeared in the Proceedings of the 1991 ACM-SIAM Symposium on Discrete Algorithms.

†Courant Institute, New York University, 251 Mercer Street, New York, New York 10012 (cole@cs.nyu.edu).

Another approach to improving the bound on the number of comparisons for the BM algorithm is to modify the algorithm so that it no longer necessarily compares characters of the pattern in consecutive right-to-left order. Such a modification was given in [2]; they thereby obtained a bound of $2n - m + 1$ character comparisons, at the cost of a more complex control structure. Crochemore et al. [8] showed that remembering just the most recently matched portion of the text reduces the upper bound of BM from $3n$ to $2n$ comparisons.

Recently algorithms that combine the left-to-right sweep of the KMP algorithm with the right-to-left sweep of the BM algorithm have been studied. In particular, Galil and Giancarlo analyzed and modified a string matching algorithm designed by Colussi [11]; they showed it makes at most $\frac{4}{3}n$ comparisons. In fact, [11] give this bound in a sharper form as a function of the period z of the pattern; the bound becomes $n + (n - m)\min\{\frac{1}{3}, (\min\{z, m - z\} + 2)/2m\}$. Very recently, almost tight bounds on the comparison complexity of string matching algorithms have been given by Cole et al. [9], [10]; they give an upper bound of $n(1 + 8/(3(m + 1)))$ character comparisons and a lower bound of $n + 2(n - m)/(m + 3)$ character comparisons for infinitely many m.

However, their lower bound does not appear to cover algorithms of the BM type, which can take advantage of a finite alphabet and perform a bad character shift (see §2). The bad character shift can contribute substantially to the expected sublinear behavior of the BM algorithm. Likewise, the algorithms of Colussi et al. and Cole and Hariharan do not incorporate a shift of this type, nor is it clear whether these algorithms can be modified to incorporate shifts of this type. So while the worst-case bound of these algorithms is better than that for the BM algorithm, it is still of interest to understand the behavior of the BM algorithm.

The remainder of the paper is organized as follows. In §2, we briefly review the BM algorithm. In §3 we show the lower bound of roughly $3n$ comparisons. In §4 we prove a number of lemmas needed for both upper bound results. In §5 we complete the proof of the $4n$ upper bound for nonperiodic patterns; in §6 we extend this upper bound to an upper bound of roughly $3n$ comparisons. In §7 we extend the results to periodic patterns.

2. The BM algorithm: A review. We first describe BM-type algorithms and then specify the BM algorithm itself. To find occurrences of the pattern in the text, a BM-type algorithm tests whether given substrings, t', of the text of length m match the pattern; each such test is called an *attempted match*. An attempted match is performed as follows. The characters of the pattern are compared one by one, in right-to-left order, with the corresponding characters of substring t' until either a mismatch is found or the match is complete. When an attempted match completes (either by a mismatch or by finding a match) the pattern is shifted to the right, ideally by the maximum distance consistent with not missing any potential matches; the actual shift may be smaller than this maximum. The shift is determined by a shift function associated with the algorithm. Following the shift, another attempted match is performed. This procedure is continued until the pattern is shifted beyond the right end of the text. The initial attempted match is with the leftmost m characters of the text.

The BM algorithm uses as its shift function the maximum of the following two shift functions. If there is a mismatch, the first shift function, the *bad character shift*, implicitly provides the location of the rightmost character, c, in the pattern, if any, that matches the mismatched character in the text. If c is present the shift specified by the bad character shift would cause c to become aligned with the mismatched text character. If c is not present the shift aligns the leftmost character of the pattern with the text character immediately to the right of the mismatched text character. The second shift function, the *good suffix shift*, is illustrated in Fig. 1; it specifies the smallest shift such that the shifted pattern matches the unshifted pattern on all the characters that were successfully matched, and fails to match the unshifted pattern on the mismatched character. If there is no such shift then the smallest shift that causes a prefix of the shifted pattern to match a suffix of the matched characters is specified. If there is no shift of this type either, then a shift of length m is specified.

Variants of the BM algorithm use the bad character shift only for a mismatch at the first character, or indeed omit its use altogether. Whichever is done, our upper and lower bounds apply. The presence of the bad character shift is desirable, however, since it helps ensure a sublinear behavior in practice. The only difficulty is that the proof of the upper bound becomes more involved, although the essentials are unchanged.

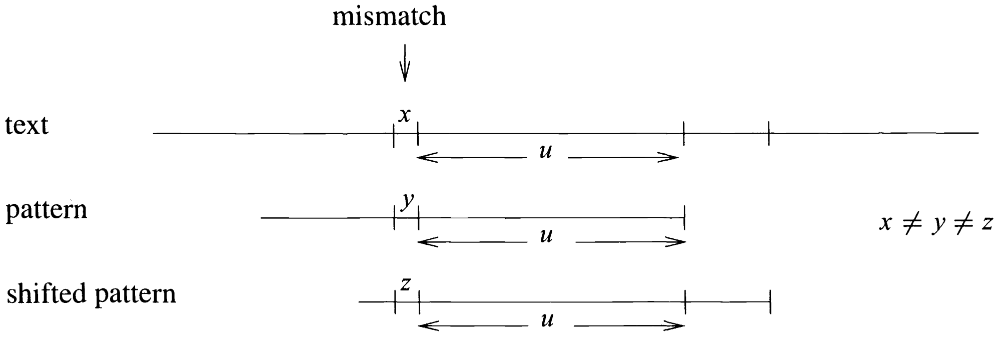


FIG. 1. *The good suffix shift.*

Remark. The terms bad character shift and good suffix shift are due to [12].

3. The lower bound. We give example patterns p, of length $m = 2k - 1$, and texts t, which demonstrate the lower bound of $3n(1 - o(1))$ comparisons (as $\frac{n}{m} \to \infty$ and $m \to \infty$).

The basic idea is to have frequent shifts by k on attempted matches comprising $2k - 1$ comparisons. In addition, to have roughly three comparisons per text character, we also seek to have short shifts on attempted matches of roughly k comparisons.

In particular, we choose $p = a^{k-1}ba^{k-1}$ and $t = a^{k-1}(aba^{k-1})^{\lambda}$. Let $v' = aba^{k-1}$.

Consider a situation in which the b in the pattern is aligned with the left end of a v' in the text, denoted $\bar{v}'$. (See Fig. 2.) The attempted match will be of length $k - 1$ and it will cause a shift of length 1. The next attempted match will have right(p) aligned with right($\bar{v}'$). The attempted match will be of length $2k - 1$ and it will result in a shift of length k. We return to the first situation. We note that t has been chosen so that the initial situation is the first situation. Thus we have shown the results given in Fig. 2.

text	···	a	b	a	a	a	b	a	a	a	b	a	a	···	
pattern				a	a	b	*a*	a							mismatch at *a*
first shift					a	a	b	a	a						match
second shift								a	a	b	a	a			iterate

FIG. 2. *The lower bound for* $k = 3$.

THEOREM 3.1. *There exist patterns of length* $m = 2k - 1$ *and texts of length* $n = \lambda(k + 1) + (k - 1)$, *for any integers* $k \geq 2$ *and* $\lambda \geq 1$, *for which the* BM *string matching algorithm performs*

$$\frac{3k-2}{k+1}(n-k+1) = (n - \frac{m-1}{2})(3 - \frac{10}{m+3})$$

comparisons. This is $3n(1-o(1))$ *comparisons as* $\frac{n}{m} \to \infty$ *and* $m \to \infty$.

For even length patterns, with $p = a^{k-2}ba^{k-1}$, for $k \geq 2$, and t one character shorter at the left end, we obtain a similar bound of

$$\frac{3k-3}{k+1}(n-k+2) = (n - \frac{m-2}{2})(3 - \frac{12}{m+4})$$

comparisons.

4. Preliminary lemmas.

DEFINITION 4.1. *A string u is* periodic *if $u = wv^k$, where w is a proper suffix of v and $k \geq 2$. u is 3-periodic if $k \geq 3$. If $w = \epsilon$, u is said to be* cyclic. *Also, u is said to be, respectively,* periodic, cyclic *in v. It is convenient to extend this terminology to allow the wording u is periodic (respectively, cyclic) in v to include the case $k = 1$; no ambiguity will result. u is* primitive *if it is not cyclic. If $u = v^k$, with v primitive, $k > 1$, v is called the* generator *of u. If $u = wv^k$, with v primitive, w a proper suffix of v, $k \geq 2$, v is called the* core *of u.*

The following lemma and corollary are well known (see [22], for example).

LEMMA 4.2. *Let x and y be two nonempty strings. If $xy = yx$ then there is a string z such that both x and y are cyclic in z.*

Proof. The proof is by induction on $|x| + |y|$. If $|x| = |y|$, then take $z = x$ $(= y)$; the result follows. Otherwise, without loss of generality, suppose that $|x| < |y|$. Then x is a prefix of y, so $y = xy_1$. Note that $y_1 \neq \epsilon$. Substituting gives $xxy_1 = xy_1x$, i.e., $xy_1 = y_1x$. The result now follows by induction. □

COROLLARY 4.3. *Suppose that v is a proper cyclic shift of w. If $v = w$, v is cyclic.*

Proof. Since v is a proper cyclic shift of w, we can write $w = xy$ and $v = yx$, where $x, y \neq \epsilon$. By Lemma 4.2, there is a string z with $x = z^i$ and $y = z^j$ for some $i, j \geq 1$. So $v = z^k$, for some $k \geq 2$, i.e., v is cyclic. □

LEMMA 4.4. *Let w be a string and let $s \leq |w|$ be an integer. Suppose that the prefix of w of length $|w| - s$ matches the suffix of $|w|$ of length $|w| - s$. Then w is periodic in u, the length s suffix of w.*

Proof. Consider two instances of w, overlapping in $|w| - s$ characters, as shown in Fig. 3. Denote the leftmost instance by w_L and the rightmost by w_R. Then $w_L = z_Lu$, $w_R = z_Ru$, for some identical strings z_L, z_R. If $|z_R| < |u|$, z_R is a suffix of u and the result is immediate. Otherwise, as w_L and w_R match on their aligned portions, w_R has u as its length s suffix; the result then follows by an inductive argument applied to z_L and z_R. □

The analysis will focus on one attempted match at a time; the attempted match AM being analyzed is called the *current attempted match*. We introduce the following notation and definitions.

DEFINITION 4.5. *t denotes the portion of text matched by AM. s denotes the length of the good suffix shift for AM. u denotes the suffix of the pattern of length s. Suppose that $u = v^k$, $k \geq 1$, where v is primitive. An attempted match AM', which precedes AM and begins by comparing a character of t, is called an* early attempted match *(with respect to AM). Throughout the paper, t and v are defined with respect to the current attempted match. w_r denotes the rightmost character of w.* right(w, z) *denotes the rightmost substring of w equal to z. w_l and* left(w, z) *are defined analogously. Let w and z be two overlapping strings; character c in string w is said to be* aligned *with z if it is aligned with some character in z.*

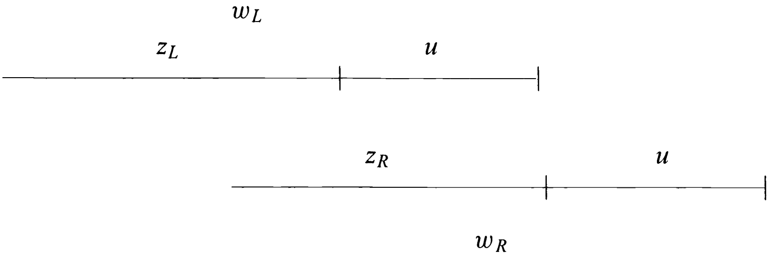


FIG. 3. *Overlapping instances of u.*

LEMMA 4.6. (i) *t and the suffix w of p of length* $\min\{|t| + |u|, |p|\}$ *are both periodic in u and v.*

(ii) *Suppose $|t| + |u| < |p|$. Then $p[m - |t| - |u|] \neq p[m - |t| - |u| + |v|]$ (i.e., the character immediately to the left of the length $|t| + |u|$ suffix of p is not equal to the character in p distance $|v|$ to its right. In addition, the character immediately to the left of t differs from $t[v]$.*

Proof. See Fig. 4. Because the good suffix shift has length $|u|$, w obeys the conditions of Lemma 4.4; thus w is periodic in u. Because u is periodic in v, result (i) for w follows. And because t is a suffix of w, result (i) for t follows also.

Result (ii) is immediate from the definition of the good suffix shift. □

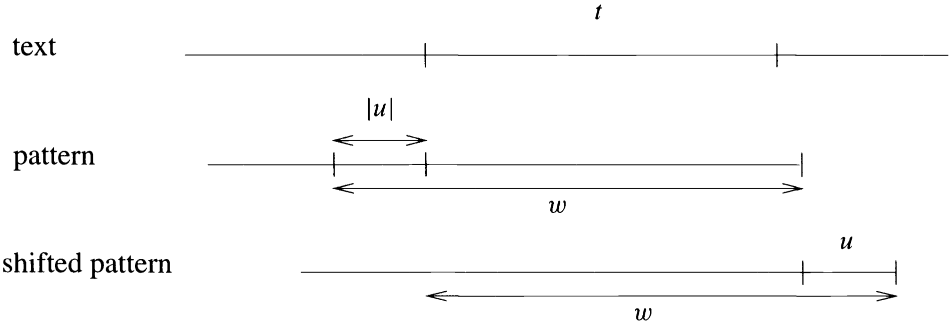


FIG. 4. *t is periodic.*

LEMMA 4.7. *Let AM_1 be an early attempted match. Suppose that $|p| > |t|$. Then p_r is not aligned with v_r for any substring v in t.*

Proof. See Fig. 5. Suppose that the pattern were so aligned. Let t_a be the text character immediately to the left of t_l. By Lemma 4.6(i), the suffix of p of length $\min\{|t| + |u|, |p|\}$ is periodic in u; thus the character in p aligned with t_a during AM_1 is identical to the character aligned with t_a during the current attempted match. Consequently, the good suffix shift would shift p_r to distance s to the right of t_r, contradicting the fact that AM_1 is an early attempted match. (Any shorter shift by a multiple of $|v|$ would place the same character in the mismatch location; this is not a good suffix shift. Any other shorter shift causes a proper cyclic shift of v (in p) to be aligned with an instance of v in t, and this cannot be a good suffix shift by Corollary 4.3, since v is primitive.) □

LEMMA 4.8. *Let AM_1 be an early attempted match. Suppose that either*

(i) $|p| > |t|$, *or*

(ii) $|p| = |t|$ *and p_r is not aligned with $\bar{v}_r$ for any instance $\bar{v}$ of v in t.*

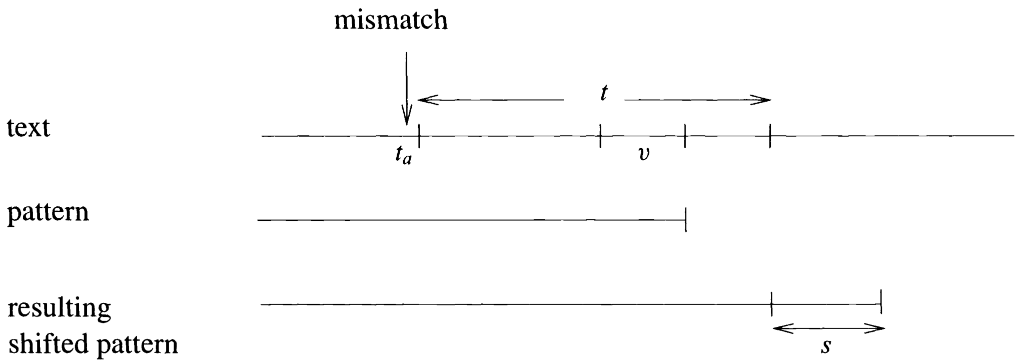


FIG. 5. *Alignment of p.*

Then AM_1 performs at most $|v|$ comparisons with characters of t. Further, if there are $|v|$ such comparisons, the last comparison is a mismatch.

Proof. p_r is not aligned with v_r for any substring v in t (by Lemma 4.7 for $|p| > |t|$ and by assumption for $|p| = |t|$). Thus if $|v|$ characters of t were matched, by Corollary 4.3, v would be cyclic, contrary to its definition. □

LEMMA 4.9. *Let AM_1 be an early attempted match. If $|p| > |t|$, p_r is either aligned with a character in* right(v, t) *or with one of the leftmost $|v| - 1$ characters in t. This remains true for $|p| = |t|$, on the condition that p_r is never aligned with $\bar{v}_r$ for any instance $\bar{v}$ of v in t apart from the rightmost instance.*

Proof. See Fig. 6. If p_r is elsewhere, the first $|v|$ comparisons, if there were that many, would be with characters in t. Then by Lemma 4.8, there must be a mismatch on or before the $|v|$th comparison, i.e., there are at most $|v|$ comparisons. Let $\bar{v}$ denote the substring v of t with which p_r is aligned. But then the shift at most moves p_r to $\bar{v}_r$. (This shift would produce a match with all the text characters compared in the attempted match and so is at least as long as the actual match.) In fact, if p_r is shifted less far, eventually, following a sequence of such attempted matches, p_r is aligned with $\bar{v}_r$. But this contradicts Lemma 4.7. □

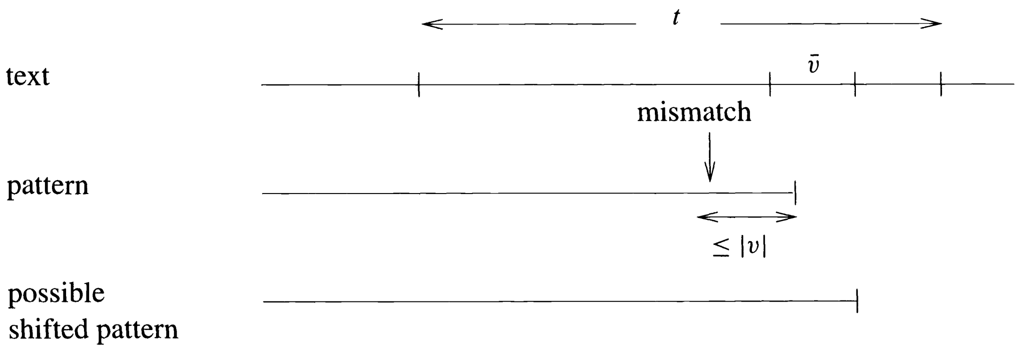


FIG. 6. *Location of p_r.*

5. A simple upper bound: 4n comparisons.

LEMMA 5.1. *If $|p| > |t|$, prior to the current attempted match at most $3|v| - 3$ characters of t have been compared.*

Proof. See Fig. 7. By Lemma 4.9, in the early attempted matches, p_r is either aligned with one of the rightmost $|v|$ characters of t or with one of the leftmost $|v| - 1$ characters of t.

By Lemma 4.8, each of the early attempted matches performs at most $|v|$ comparisons. Since t_r is not compared prior to the current attempted match, the lemma follows. □

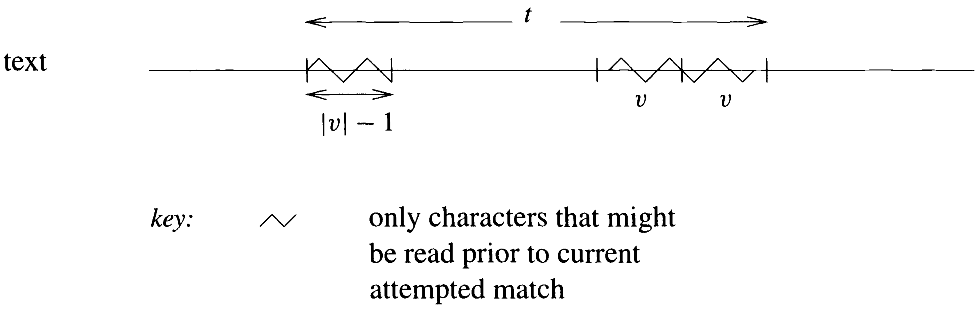


FIG. 7. *Characters already read.*

THEOREM 5.2. *The* BM *pattern string algorithm performs at most* $4n$ *comparisons when matching a non-*3*-periodic pattern of length m against a text of length n.*

Proof. We will show that the number of comparisons performed in each attempted match AM is bounded as follows.

Attempted match bound. The attempted match bound equals the number of text characters compared for the first time during AM plus three times the length of the ensuing shift.

It then follows that the total number of character comparisons is at most $4n$.

Consider AM, the current attempted match. If $|t| \leq 3s$, because the number of characters compared is $|t|$ or $|t|+1$, the attempted match bound holds for AM (since t_r was compared for the first time during the current attempted match).

So suppose that $|t| > 3s$. Since p is not 3-periodic, $|p| > |t|$. By Lemma 5.1, the number of previously uncompared characters compared by the current attempted match is at least $|t| - (3|v| - 3) \geq (|t| + 3) - 3s$; because the number of characters compared by AM is $|t| + 1$, the attempted match bound holds in this case too. □

6. An upper bound of roughly $3n$ comparisons. The $3n$ upper bound is proved using an amortized analysis, an approach for analyzing algorithms due to Sleator and Tarjan [24]. The method is as follows. Following the ith shift performed by the algorithm a potential ϕ_i is associated with the current state of the algorithm. ϕ_0 denotes the initial potential, before any shifts. Let t_i be the number of comparisons performed by the ith attempted match (i.e., between the $(i-1)$th and the ith shifts). Then the amortized cost of the ith attempted match, in comparisons, is defined to be $t_i + \phi_{i+1} - \phi_i$. The reason for using amortized costs is that with a well-chosen potential function, they can be easier to bound than the actual costs.

The total number of comparisons performed by the algorithm is given by

$$\sum_{i\geq 1} t_i = \sum_{i\geq 1} (a_i - \phi_{i+1} + \phi_i) = \sum_{i\geq 1} a_i - \phi_f + \phi_0,$$

where ϕ_f is the final potential, the potential at the end of the algorithm.

The $3n$ bound is proved using the following potential function.

$\phi = 2 \times$ the number of still-readable characters $+$ the number of unmarked characters.

A character is readable if it is not yet to the left of the left end of the pattern. For the purposes of analysis characters may be marked and then unmarked, possibly repeatedly. Initially every character is unmarked.

Clearly $\phi_0 = 3n$ and $\phi_f \geq 0$. We will show that the amortized cost (in comparisons) of each attempted match is at most -1. Thus the total number of comparisons performed is bounded by $3n$. In fact, the negative bound on the cost of each attempted match leads to a slightly tighter bound, as we will see.

Now we turn to the analysis of the BM algorithm. As in §5, we are assuming that the pattern is non-3-periodic.

As in §5, we consider a current attempted match that matches a substring t of the text and either has $|p| = |t|$ or mismatches at character t_a immediately to the left of t_l. We also consider earlier attempted matches in which p_r is aligned with t. Let s be the distance shifted in the current attempted match. There are a number of cases to consider; in all the cases we show an amortized cost of -1 or better for the current attempted match.

Before proceeding with the analysis, we specify the rules for unmarking text characters. When an unmarked character is read it becomes marked. However, a marked character can be unmarked again as follows. Whenever an attempted match would have an amortized cost less than -2, if the shift made is due to the good suffix shift, then the leftmost characters compared in the current attempted match are unmarked so as to increase the amortized cost to -2 (or until all the characters read in the current attempted match are unmarked, whichever occurs sooner). Actually, as we see below in Case 1, there is a special case for shifts of length 1. If the shift of length s is due to the bad character shift and the good suffix shift had length $s' < s$, then the same characters are unmarked as for a good suffix shift of length s'. In addition, the rightmost $\min\{2(s - s'), |t| + 1\}$ characters compared are unmarked.

Case 1. $s = 1$. Let t' be the text read by the current attempted match. Then $t' = ba^j$ for some $j \geq 0$, where $a \neq b$. We claim that the characters of t' had not been read previously. This would give an overall amortized cost of -2 for the attempted match, except that we unmark t'_r, giving an amortized cost of -1. It remains to demonstrate the claim. If there had been an earlier attempted match with p_r aligned with t', then the mismatch would have been at character b of t' and the resulting shift would have moved p_r beyond t'_r; therefore there was no such earlier attempted match, i.e., all of t' was unmarked immediately prior to the current attempted match.

Remark 6.1. Henceforth we can assume that at each attempted match the rightmost two characters compared (if at least two are compared) are unmarked.

Case 2. $|t| < 2s$. The decrease in potential is at least $2s + 2$. The cost of the current attempted match is $|t| + 1$. So the amortized cost of the current attempted match is at most -2.

Case 3. $|t| \geq 2s$ and $s > 1$. Then by Lemma 4.6(i), $t = wv^k$, for some $k \geq 2$, where w is a proper suffix of v, v is primitive, and s is an integer multiple of $|v|$. A little more notation is helpful. Let w_L, v_L, v_{R1}, v_{R2} denote, respectively, left(w, t), left(v, t), right(v, t), and the second rightmost v in t. It may be that $v_L = v_{R2}$. (See Fig. 8.) There are five subcases to consider.

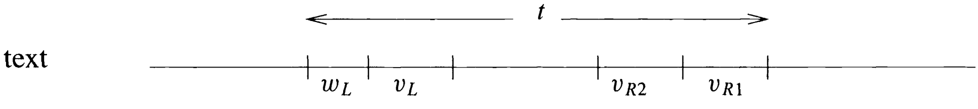


FIG. 8. *Further notation.*

Case 3.0. There is no early attempted match.

Case 3.1. AM_1, the first early attempted match, has p_r aligned with a character of v_{R1} other than $(v_{R1})_r$.

Case 3.2. AM_1 is an early attempted match which results in a shift by distance greater than $|p| - |v|$.

Case 3.3. AM_1 is an early attempted match in which p_r is aligned with w_L and following which p_r becomes aligned with v_{R1}. In addition, the resulting shift has length at most $|p| - |v|$.

Case 3.4. AM_1 is an early attempted match in which p_r is aligned with v_L but not with $(v_L)_r$, and following which p_r becomes aligned with v_{R1}. In addition, the resulting shift has length at most $|p| - |v|$.

Case 3.5. AM_1, an early attempted match, had p_r aligned with $(v_L)_r$.

By Lemma 4.9, if $|p| > |t|$, either case 3.0 applies or there is an early attempted match falling into one of cases 3.1–3.4. If $|p| = |t|$, $p = wvv$, since p is not 3-periodic. If none of cases 3.0–3.4 apply, the only remaining possibility is given by case 3.5. We analyze each of the cases in turn.

Case 3.0. There is no early attempted match. Then the current attempted match has an amortized cost of at most $-2s + 1 \leq -3 \leq -2$.

Case 3.1. AM, the first early attempted match, has p_r aligned with a character of v_{R1} other than $(v_{R1})_r$. Then at most $2|v| - 3$ characters of t will have been read and remain marked prior to the current attempted match (because among the characters of v_{R1} the rightmost two are unmarked, and by Lemma 4.8, besides characters of v_{R1}, only the $|v| - 1$ rightmost characters of v_{R2} can have been read). Thus the amortized cost of the current attempted match is at most -2.

In the remaining subcases, the analysis will focus on the early attempted match that causes p_r to become aligned with v_{R1}.

Before considering the next case, we prove several lemmas.

LEMMA 6.1. *Let AM_1 be an early attempted match for which the good suffix shift has length $s_1 \leq |p| - |v|$.*

(i) *If AM_1 matches at least $|v|$ characters, then s_1 is an integer multiple of $|v|$.*

(ii) *If AM_1 matches fewer than $|v|$ characters, then $s_1 < |v|$.*

Proof. First note that the suffix of p of length $s_1 + |v|$ is periodic in v. This follows by Lemma 4.6(i), since $s_1 < |t|$ and thus $s_1 + |v| \leq \min\{|p|, |t| + |u|\}$.

Now we prove (i). Let $\bar{v}$ be the string in the text which matches right(v, p) during attempted match AM_1. Following the shift, the substring $\bar{\bar{v}}$ of p aligned with $\bar{v}$ must be identical to the pattern v (since $s_1 \leq |p| - |v|$, $\bar{\bar{v}}$ exists). Furthermore, $\bar{\bar{v}}$ is part of the suffix of p of length $s_1 + |v|$, and this suffix is periodic in v. Thus $\bar{\bar{v}}$ is a cyclic shift of v. By Corollary 4.3, because v is primitive this cyclic shift must be the trivial shift; hence s_1 is an integer multiple of $|v|$.

We turn to (ii). Suppose that $s_1 > |v|$. Further suppose that c characters are compared by AM_1. (See Fig. 9.) Then a shift by $s_1 - |v|$ would also be a legal shift, in that the characters of the pattern aligned with the c text characters compared by AM_1 are identical in a shift by s_1 and a shift by $s_1 - |v|$ (because the portion of p periodic in v has length at least $s_1 + |v|$). Thus $s_1 \leq |v|$. But a shift by $|v|$ is not possible because this would replace the mismatched character of the pattern by the same character, which is not a good suffix shift. Thus $s_1 < |v|$. □

LEMMA 6.2. *Let AM_1 be an early attempted match that shifts p so that p_r is aligned with a character of v_{R1} other than $(v_{R1})_r$. Suppose AM_1 uses the bad character shift. If at least $|v|$ characters are matched by AM_1, then the good suffix shift for AM_1 also aligns p_r with v_{R1}.*

Proof. By Lemma 6.1, the shift due to the good suffix shift has length at least $|v|$. If $|p| > |t|$, by Lemma 4.9, p_r becomes aligned with a character of v_{R1}. If $|p| = |t|$, p is periodic in v and the bad character shift moves p so that p_l is to the immediate right of the mismatched text character. (If not the shift would have length less than $|v|$.) Similarly,

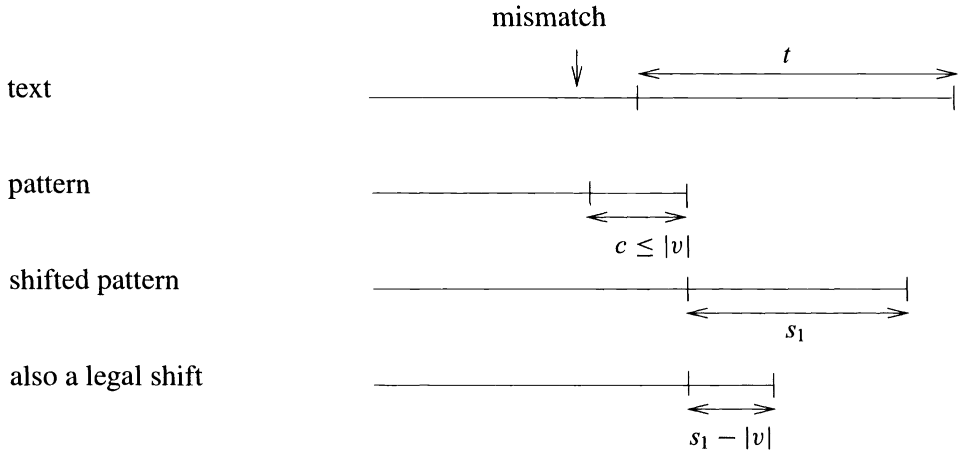


FIG. 9. *Case* (ii): *what if* $s_1 > |v|$.

the good suffix shift would shift p so that the mismatched text character was to the left of p_l. (If not p would overlap the matched characters after the shift, and its length would therefore be a multiple of $|v|$; but then identical characters would be aligned with the mismatched text character before and after the shift, which would not be a good suffix shift.) But then the good suffix shift would be at least as long as the bad character shift, contrary to assumption. □

LEMMA 6.3. *Let AM_1 be an early attempted match that uses the bad character shift and compares at most $|v|$ characters. Then its shift has length either greater than $|p| - |v|$ or less than $|v|$.*

Proof. Suppose that the shift has length at most $|p| - |v|$. Then after the shift a pattern character p_a is aligned with the mismatched text character; consequently, p_a must be among the rightmost $|t| + |v|$ characters of p. But because the suffix of p of length $\min\{|t| + |v|, |p|\}$ is periodic in v, by the definition of the bad character shift p_a is among the rightmost $|v|$ characters in p. Thus in this case the shift has length less than $|v|$. □

Case 3.2. AM_1 is an early attempted match which results in a shift by distance more than $|p| - |v|$. Then all but the rightmost $|v| - 1$ characters shifted over remain unmarked until the current attempted match, by Lemma 4.8. Let w' be the suffix of p overlapping t at the start of AM_1. $|w'| + |p| - |v| + 1 \leq |t| \leq |p|$; hence $|w'| \leq |v| - 1$. If $|w'| = |v| - 1$ then $|w'|$ characters of t are marked at the time of the current attempted match; thus is has amortized cost at most -2. If $|w'| = |v| - 2$ then after AM_1 p_r is aligned with one of the two rightmost characters in v_{R1}; the next and last early attempted match, if any, compares at most $|v|$ characters. Thus for the current attempted match there are at most $|w'| + |v| - 1$ marked characters in t; hence it has amortized cost at most $(2|v| - 3 + 1) - 2s \leq -2$. If $|w'| \leq |v| - 3$ there are at most $[(|v| - |w'| - 1) - 2] + |v| + |w'| \leq 2|v| - 3$ marked characters for the current attempted match. (The first term is the number of characters in t to the right of p_r after the shift due to AM_1; -2 is due to the fact that the rightmost two characters in t are unmarked at the start of the current attempted match; $|v|$ is the bound due to Lemma 4.8 on the number of characters that become marked that are aligned with or to the left of p_r after the shift due to AM_1; and $|w'|$ is for the characters marked by or before AM_1.) This yields an amortized cost of at most -2 for the current attempted match.

Case 3.3. AM_1 is an early attempted match in which p_r is aligned with w_L and following which p_r becomes aligned with v_{R1}. In addition, the resulting shift has length at most $|p| - |v|$.

We show that at most $2|v| - 3$ characters of t are read in early attempted matches; it follows that the current attempted match has amortized cost at most -2.

First we show that AM_1 matches at least $|v|$ characters. Clearly the shift has length greater than $|v|$ as p_r shifts over the whole of v_L. If the shift is due to the good suffix shift, then the claim is immediate from Lemma 6.1(ii). Likewise, if the shift is due to the bad character shift, then the claim is immediate from Lemma 6.3.

Suppose that at the start of AM_1 p_r is aligned with the rth rightmost character of w_L. We show that following AM_1, p_r is aligned with or to the right of the rth rightmost character of v_{R1}. Let s_1 be the length of the good suffix shift due to AM_1. By Lemma 6.2, a shift by distance s_1 following attempted match AM_1 would align p_r with v_{R1}. By Lemma 6.1(i), s_1 is an integer multiple of v.

By Lemma 4.8, at most $|v| - 1$ characters of the text immediately to the left of the rth rightmost character of v_{R1} can be read prior to the current attempted match, and then only if $r > 1$. Thus the number of characters of t that can be marked prior to the current attempted match is bounded as follows: at most $|w| - r + 1$ characters of w_L, plus at most $\max\{(r-2) + (|v| - 1), 0\} = (r-2) + (|v| - 1)$ characters at the right end of t (recall that the two rightmost characters of v_{R1} are unmarked). This is a total of $|w| + |v| - 2 \leq 2|v| - 3$ characters. Hence the current attempted match has amortized cost at most -2.

Case 3.4. AM_1 is an early attempted match in which p_r is aligned with v_L but not $(v_L)_r$, and following which p_r becomes aligned with v_{R1}. In addition, the resulting shift has length $s_1 \leq |p| - |v|$. The characters matched by AM_1 include at least all the characters up to t_l. (If not a shift that aligns p_r and $(v_L)_r$ would leave all the compared characters matched and hence would be as long as the actual shift.) There are two subcases depending on the number of characters matched by AM_1.

Case 3.4.1. At least $|v|$ characters are matched.

The following two lemmas will be useful.

LEMMA 6.4. *Suppose that an early attempted match AM_2 starts with suffix x of p overlapping v_{R1}. Suppose further that the maximum length suffix of v periodic in x is $\tilde{w}x$. Finally, suppose that x is also a prefix of v. Then*

(i) $|\tilde{w}x| < |v|$.

(ii) *AM_2 mismatches at the $(|\tilde{w}x| + 1)$th character compared.*

(iii) *Suppose AM_2 produces a good suffix shift of length s_2. Let $\bar{x}$ be the generator of x. Then $s_2 > |\tilde{w}x| - |\bar{x}| \geq |\tilde{w}|$.*

Proof. See Fig. 10. By Lemma 4.8, AM_2 compares at most $|v|$ characters. Since v and p both have suffix $\tilde{w}x$ and v_{R1} has prefix x, it is clear that AM_2 matches the first $(|\tilde{w}x|)$ characters compared; hence, $|\tilde{w}x| < |v|$. Since $\tilde{w}x$ is the longest suffix of v, and hence of p, periodic in x it follows that AM_2 mismatches at the $(|\tilde{w}x| + 1)$th character compared.

Let z be the suffix of p of length s_2. Suppose, for a contradiction, that $|z| \leq |\tilde{w}x| - |\bar{x}|$. By Lemma 4.4, the suffix of p of length $|z\tilde{w}x|$ is periodic in z as the good suffix shift due to AM_2 causes a match on the characters matched by AM_2. Consider the suffix of length $|\bar{x}|$ of the second rightmost instance of z in p; it lies within the suffix $\tilde{w}x$ of p, which is periodic in $\bar{x}$. By Corollary 4.3, any matching cyclic shift of this suffix of z is a trivial cyclic shift; consequently, the length $|z|$ suffix of $\tilde{w}x$, and hence z itself, are cyclic in $\bar{x}$. Since v is also a suffix of p and $|\tilde{w}| + 1 \leq |v|$, the rightmost $|\tilde{w}| + 1$ characters of v are periodic in $\bar{x}$, thus there is no mismatch by AM_2 at the location assumed (the $(|\tilde{w}x| + 1)$th character compared by AM_2). This is a contradiction. Consequently $s_2 = |z| > |\tilde{w}x| - |\bar{x}|$. □

Case 3.4.1.A. AM_1 uses the good suffix shift. Let x be the suffix of p overlapping v_L before attempted match AM_1, let y be the suffix of p overlapping v_{R1} after attempted match AM_1, and let $\tilde{w}x$ be the longest suffix of v periodic in x. In turn, we show:

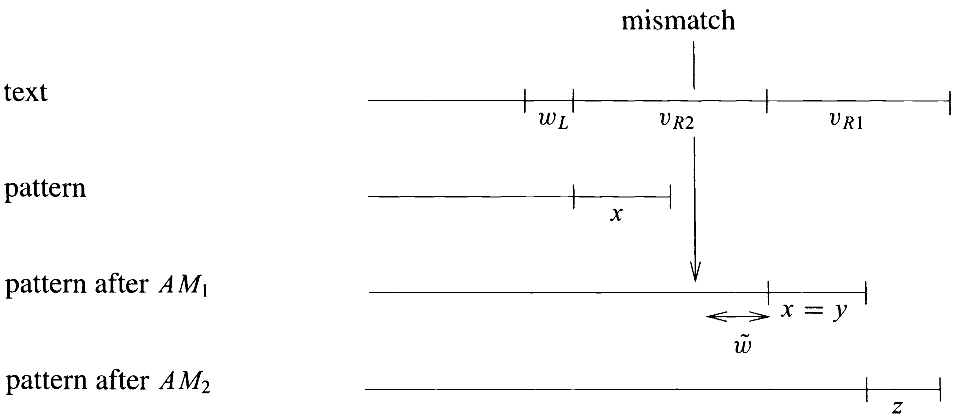


FIG. 10. *Attempted match AM_2.*

(i) $|x| = |y|$.

(ii) The attempted match AM_2, immediately following AM_1, mismatches at the $(|\tilde{w}x|+1)$th character compared.

(iii) Let the good suffix shift due to AM_2 have length s_2. Then $s_2 > |\tilde{w}| \geq |w|$.

(iv) The amortized cost of the current attempted match is at most -2.

By Lemma 6.1(i), s_1 is an integer multiple of $|v|$. Thus $x = y$. This shows (i). (ii) and the first part of (iii) are immediate from Lemma 6.4. To see the remainder of (iii), consider the characters matched by AM_1. It follows that v has suffix wx; in addition, by Lemma 4.4, wx is periodic in x. Thus $|\tilde{w}| \geq |w|$.

Prior to unmarking read nodes, the amortized cost of AM_2 would be at most $-2s_2$, since all the characters compared by AM_2 were previously unread (for $|\tilde{w}x| < |v|$ by Lemma 6.4(i)). Hence $\min\{2s_2 - 2, |\tilde{w}x| + 1\} \geq |\tilde{w}|$ characters are unmarked. Following this unmarking, the $|\tilde{w}| \geq |w|$ characters immediately to the right of left(x, v_{R2}) are unmarked; by Lemma 4.8, these characters are not read again prior to the current attempted match. So the number of marked characters in t at the time of the current attempted match is bounded by the sum of:

(i) $(|v| - 2)$ characters in v_{R1},

(ii) If $v_{R2} = v_L$, $(|v| - |w|)$ characters in v_{R2}, and otherwise $|x|$ characters in v_L and $(|v| - |w| - |x|)$ characters in v_{R2},

(iii) $|w|$ characters in w_L.

This is a total of at most $2|v| - 2$ marked characters. Hence the amortized cost of the current attempted match is at most -1. (This can be improved to -2 by a more elaborate argument.)

Case 3.4.1B. AM_1 uses the bad character shift. Let x denote the overlap of p and v_L prior to attempted match AM_1. Let AM_2 be the first subsequent early attempted match that compares a character of v_{R2}, if any.

We observe that the bad character shift overlaps v_{R1} by more than $|x|$ characters. By Lemma 6.2, the good suffix shift of AM_1 aligns p_r with v_{R1}. Since the length of the good suffix shift, in this case, is a multiple of $|v|$ (by Lemma 6.1(i)), following the good suffix shift p would overlap v_{R1} by $|x|$ characters. Since the bad character shift is used, it results in a larger overlap.

There are three subcases.

Subcase (i). AM_2 does not exist, or in AM_2, p_r is to the right of the leftmost $|wx|$ characters of v_{R1}. Then the number of marked characters in t at the time of the current attempted match is at most $(|v| - 1 - |wx|) + (|v| - 2) + |w| + |x| \leq 2|v| - 3$ ($|v| - 2$ bounds the number of marked characters in v_{R1}, $|v| - 1 - |wx|$ bounds the number in v_{R2} marked after the shift by AM_1, and $|w| + |x|$ bounds the number marked before the shift by AM_1). Thus the amortized cost of the current attempted match is at most -2.

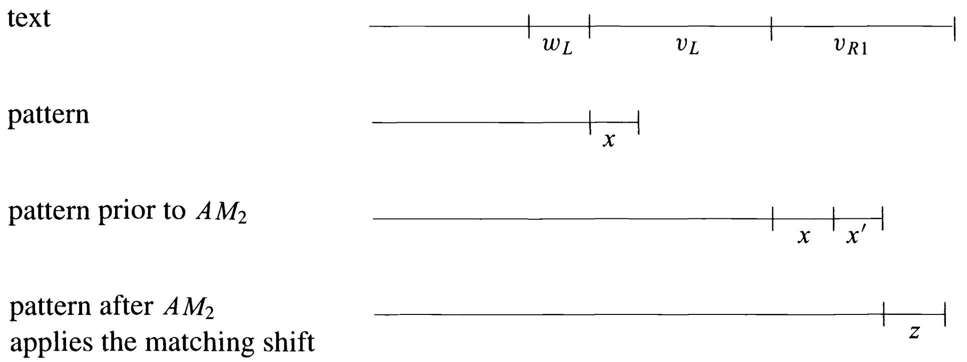


FIG. 11. Case 3.4.1B–*the bad character shift.*

Subcase (ii). p_r is aligned with one of the leftmost $|wx|$ characters in v_{R1} and $|x| \leq |v|/2$. (See Fig. 11.) Let s_2 be the length of the good suffix shift for AM_2 and let z be the suffix of p of length s_2. Let x' be the overlap of p beyond left(x, v_{R1}) at the start of attempted match AM_2; by assumption, $|x'| \leq |w|$. Let $\tilde{w}x$ be the longest suffix of v periodic in x. In turn, we show the following.

(i) Let $\bar{x}'$ be the generator for x'. x is cyclic in $\bar{x}'$.

(ii) AM_2 mismatches at the $(|\tilde{w}x| + 1)$th character compared.

(iii) $|\tilde{w}| < |z|$.

(iv) The amortized cost is at most -2.

(i) is shown as follows. Lemma 4.4 is applied to the prefix x of v_{R1} and to the suffix xx' of p, which match on their overlapping portion during AM_2; because p has suffix x also, it follows that xx' is periodic in x'. Let $\bar{x}'$ be the generator for x'. Next, by applying Lemma 4.4 to the suffix wx of p and the string w_L, which match on their overlapping portion during AM_1, it follows that wx is periodic in x. Since $|w_L| \geq |x'|$, x' is a suffix of w_L. Let $\bar{\bar{x}}'$ be the substring of p aligned with right($w_L, \bar{x}'$) during AM_1; by Corollary 4.3, $\bar{\bar{x}}'$ is the trivial cyclic shift of $\bar{x}'$. Thus x is cyclic in $\bar{x}'$. It follows that w is cyclic in $\bar{x}'$ and as p has suffix wx, $|\tilde{w}| \geq |w|$.

By Lemma 6.4, AM_2 mismatches at the $(|\tilde{w}x| + 1)$th character compared (to see this note that the longest suffix of v periodic in $\bar{x}'$ and hence xx' has length $|\tilde{w}x|$). (Note that $\tilde{w}$ has different meanings here and in the lemma.) Also by Lemma 6.4, $|z| > |\tilde{w}x| - |\bar{x}'| \geq |\tilde{w}|$.

To obtain the amortized bound we need at least $|w| + 1$ characters of v_{R2} to remain unmarked aside from left(x, v_{R2}); then at most $2|v| - 3$ characters of t will be marked at the time of the current attempted match, which yields the -2 bound on the amortized cost. Following the shift due to AM_2, the potential at hand for unmarking characters again is at least $2|z| - |x| - |x'|$; to obtain $|w| + 1$ unmarked characters in v_{R2} aside from left(x, v_{R2}), we need

$$(|v| - |x|) - (|\tilde{w}x| - |xx'| + 1) + (2|z| - |x| - |x'|) \geq |w| + 1.$$

(The first term is the number of characters in v_{R2} unmarked before AM_2; the second term is the number of characters in v_{R2} read by AM_2.) Thus it suffices that $|v| - 2|x| + 2|z| \geq 2|\tilde{w}| + 2$, which is true when $|x| \leq |v|/2$.

Subcase (iii). p_r is aligned with one of the leftmost $|wx|$ characters in v_{R1} and $|x| > |v|/2$. Let y be the generator of the suffix of p of length $|v| - |x|$. In turn, we show the following.

(i) x has core y.

(ii) AM_2 compares at most $|y|$ characters of v_{R2}.

(iii) AM_2 is the last early attempted match for the current attempted match.

(iv) The number of marked characters for the current attempted match is at most $2|v|-3$, and hence the amortized cost is at most -2.

By Lemma 4.4, v and hence x have core y. (since x, a suffix of v, matches a prefix of v). Any early attempted match AM_2, subsequent to AM_1, which matches a character of v_{R2} will have matched a suffix of p of length greater than $|x|$ ($> |y|$) and at least $|x|$ characters of v_{R1}. Also, AM_2 can compare at most $|y|$ characters of v_{R2} (this can be seen as follows: v is not cyclic in y, by assumption, and at least one character of v_{R1} is compared, thus the "y" substrings in p and v_{R2} cannot be aligned). The resulting good suffix shift will have a length of a multiple of $|y|$, since y is primitive and v is periodic in y. Furthermore, the resulting shift must align p_r and t_r (since this is the least shift by a multiple of $|y|$ that places a different character in the mismatch location).

Consequently, the number of marked characters for the current attempted match is at most $|v| + (|w| + |x|) - 2 \leq 2|v| - 3$ ($|v|$ is the number of characters compared in the attempted matches between AM_1 and the current attempted match, $|w| - |x|$ is the number of characters of t compared by AM_1, and 2 is the lower bound on the number of characters unmarked due to the bad character shift).

Case 3.4.2. Fewer than $|v|$ characters are matched. The shift is a good suffix shift. For by Lemma 6.3 a bad character shift has length less than $|v|$. Thus the bad character shift would align the rightmost instance in p of the mismatched text character with that text character, thereby leaving p_r aligned with one of the leftmost $|v|$ characters of t, contrary to the definition of Case 3.4.

Let r be the number of characters matched. Note that $r > |w|$. We show that $s_1 > r$ and then we upper bound the amortized cost by -2.

Suppose, for a contradiction, that $s_1 \leq r$. Since $s_1 < |v|$, $|y| < |x|$. (See Fig. 12.) Since a suffix of p, of length $|t|$, is periodic in v (by Lemma 4.6(i)) and the shifted pattern matches the text characters matched by AM_1, y must be a prefix of x.

Next, we show that there is a string z such that both x and y are cyclic in z. The portion $\bar{x}$ of the shifted pattern, aligned with the suffix x of p prior to the shift, must match the suffix x. Note that $\bar{x} = yv'$ where v' is a prefix of v. Prior to the shift, x_l is aligned with $(v_L)_l$, so v' is a prefix of x also. By Lemma 4.4, applied to x reversed, $x = y^i y'$ for some $i \geq 1$ where y' is a proper prefix of y. Because y is also a suffix of x (since they are both suffixes of p), using Corollary 4.3, we conclude that there is a primitive string z such that both x and y are cyclic in z.

In addition, the suffix of p of length r has core z. (Consider the portion of the pattern, following AM_1, that is aligned with the r text characters matched by AM_1; it comprises the prefix of v of length $|x| - |y|$, preceded on the left by the suffix of v of length $r - (|x| - |y|)$. Also, it matches the suffix of v of length r. By Lemma 4.4, the suffix of $|v|$ of length r is periodic in its suffix of length $|x| - |y|$; in turn, this suffix is periodic in z.) The assertion follows.

Finally, note that x is a prefix of v (consider the attempted match AM_1 and the portion of t with which x is matched).

To avoid v being cyclic in z, we need that in $\bar{v} = \text{right}(v, p)$, the suffix of length r overlaps with left$(x, \bar{v})$ by fewer than $|z|$ characters. Otherwise, by Corollary 4.3, the overlapping portion would comprise aligned instances of string z; it would follow that v was cyclic in z.

Thus we need $r+|x| < |v|+|z|$, which implies $r < |v|+|y|-|x| = s_1$; this is a contradiction. Thus the case $s_1 \leq r$ does not arise.

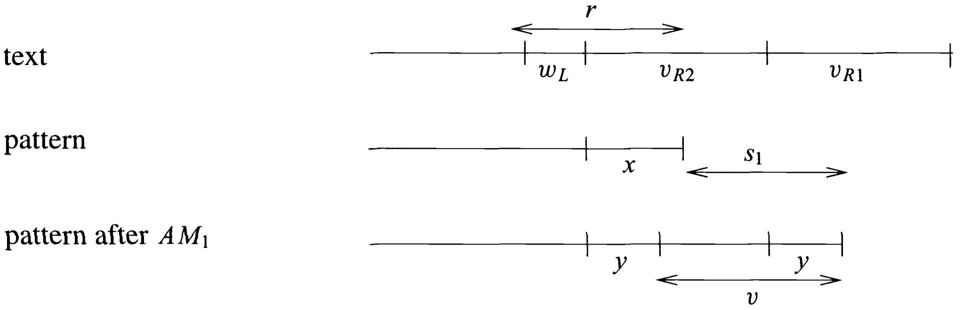


FIG. 12. *Case* 3.4.2.

Following the shift due to AM_1, up to s_1 characters can be unmarked; hence all the characters read by AM_1 are unmarked. These unmarked characters include w_L plus the character immediately to the right of w_L. By Lemma 4.8, these $|w|+1$ characters are not reread until the current attempted match. So there are at most $2|v|-3$ marked characters in t at the time of the current attempted match, and hence the amortized cost of the current attempted match is at most -2.

Case 3.5. AM_1, an early attempted match, had p_r aligned with $(v_L)_r$. Because p is not 3-periodic, $p = \tilde{w}vv$, $t = wvv$, where w is a suffix of $\tilde{w}$, which in turn is a proper suffix of v. The resulting shift has length $|v|$. Hence the number of unmarked characters for the current attempted match is at least $|v|$. Thus the amortized cost of the current attempted match is bounded by $|v|+|w|+1-2|v|$ if $|\tilde{w}| > |w|$, and by $|v|+|w|-2|v|$ otherwise; these are both at most -1.

In conclusion we have the following theorem.

THEOREM 6.5. *The* BM *string matching algorithm performs at most* $3n - 3(n-m+1)/(m+2)$ *comparisons when matching a nonperiodic pattern of length m against a text of length n.*

Proof. Clearly each pair of consecutive attempted matches, whose shifts have combined length at most $\lfloor\frac{m}{3}\rfloor + \lceil\frac{m}{3}\rceil + 1$, has amortized cost at most -2. We show that each pair of consecutive matches, whose shifts have combined length $\lfloor\frac{m}{3}\rfloor + \lceil\frac{m}{3}\rceil + 1 + s$, $s \geq 1$, has amortized cost at most $-2-s$. Let the initial attempted match result in a shift of length s'; then it has amortized cost $-2s'$ if $s' > 1$ and -1 if $s' = 1$. Thus the maximum total amortized cost is obtained if all the attempted matches form pairs with associated combined shifts of length $\lfloor\frac{m}{3}\rfloor+\lceil\frac{m}{3}\rceil+1$. This leads to a total amortized cost of at most $3n-2(n-m+1)/(\lfloor\frac{m}{3}\rfloor+\lceil\frac{m}{3}\rceil+1)$ which is bounded by $3n-3(n-m+1)/(m+2)$.

In right-to-left order, each unpaired (*long*) shift of length $\lfloor\frac{m}{3}\rfloor + 1 + r$, $r \geq 0$, is paired with the immediately preceding shift. This leaves at most one unpaired shift of length greater than $\lfloor\frac{m}{3}\rfloor + 1$, and if present it is the very first shift of the algorithm.

Consider such a pair of consecutive shifts. Let the first shift have length l. If $l+r \leq \lceil\frac{m}{3}\rceil$, then the pair of attempted matches producing these two shifts has amortized cost at most -2. If $l+r = \lceil\frac{m}{3}\rceil + s$, $s \geq 1$, the amortized cost of the second attempted match is at most $m - l - 2(\lfloor\frac{m}{3}\rfloor + 1 + r) = -1 - s - r \geq -1 - s$. Because the first attempted match has amortized cost at most -1, together the pair have amortized cost at most $-2-s$, as claimed. □

COMMENT. When matching the pattern $p = wvv$ against text $t = wv^k$, where w is the length $|v| - 1$ suffix of v and v is primitive, for $k \geq 2$, $3n - 3\frac{n+1}{m+1} - 2m + 3$ comparisons are performed, a slight improvement on the lower bound given in §3. We do not state this lower bound there, because variants of the BM algorithm avoid this case.

7. 3-periodic patterns. Suppose that the pattern, p, is of the form $p = wv^i$, where v is the core of p, w is a proper suffix of v, possibly empty, and $i \geq 3$.

One approach is to find instances of the pattern $p' = wv$. A sequence of i such instances, each separated by distance $|v|$, corresponds to an instance of pattern p. It is not hard to determine p' in $O(m)$ time. However, this is not a very attractive solution since p' can itself be periodic and thus a recursive use of this approach may be needed. Clearly this approach performs no additional comparisons with text characters over what is implied by the bound of Theorem 6.5, where the m in the theorem now denotes the length of the pattern that is actually being matched (i.e., the pattern obtained by the just-described recursive decomposition). However, additional work is needed to keep track of the sequence of patterns p' that have been found; it can be shown that over all recursive levels of such patterns, this is an additional $O(n)$ work.

Instead, we use the fact that the bound of Theorem 6.5 applies when matching pattern $p' = wvv$. Clearly there is no recursive decomposition in this case. To detect instances of pattern p, the following index j is recorded. Let $q_j = wv^j$. (See Fig. 13.) We record the largest j such that the text contains an instance $\bar{q_j}$ of the pattern q_j, where $(\bar{q_j})_r$ is aligned with p'_r.

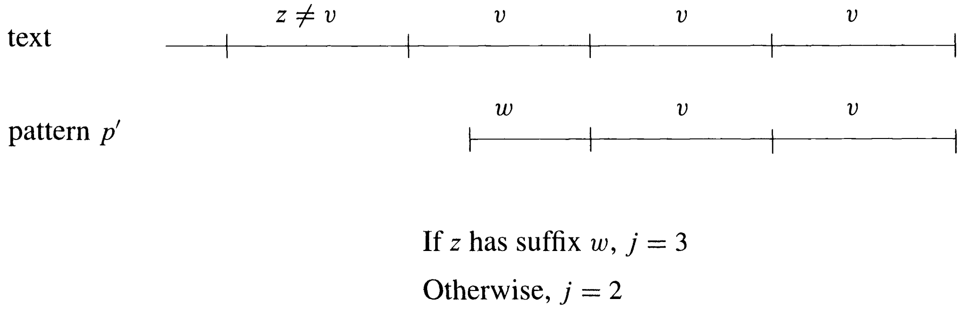


FIG. 13. *Periodic patterns.*

Following a successful match of p', j is modified as follows. If $j = i - 1$, then a match of p has been found; j is unchanged. If $j < i - 1$, then j is incremented. If the attempted match is not successful j is set to one.

In conclusion, we have the following theorem.

THEOREM 7.1. *Suppose the* BM *string matching algorithm is applied to a pattern p and a text of length n. Let $p = wv^i$, where v is the core of p. Then the* BM *algorithm performs at most* $3n - 3(n - m' + 1)/(m' + 2)$ *comparisons, where* $m' = |p|$ *if* $i \leq 2$, *and* $m' = |wv^2|$ *if* $i > 2$.

Comment. Some further slight improvements to the bound are possible. Specifically, note that $p = wv$, with v the core of p, does not have another core v', with $p = w'v'^k$, for some $k \geq 2$ if $|w| = |v| - 1$ or $|v| - 2$. Thus patterns $p = wv^k$, $k \geq 2$, with $|w| = |v| - 1$ or $|v| - 2$ can be matched by matching $p' = wv$, as in this section. This then allows the amortized bounds given earlier for cases 2 and 3 to be improved to -2 for each attempted match, giving an upper bound of roughly $3n - \frac{6n}{m}$. Further improvements are much more intricate.

Acknowledgments. I thank Rajamani Sundar for his comments. I also thank the referees for their suggestions concerning the presentation of the result.

REFERENCES

[1] A. V. AHO AND M. J. CORASICK, *Efficient string matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

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

[3] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The design and analysis of algorithms*, Addison Wesley, Reading, MA, 1973.

[4] A. AMIR, G. M. LANDAU, AND U. VISHKIN, *Efficient pattern matching with scaling*, Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1990, pp. 344–457.

[5] G. BRASSARD AND P. BRATLEY, *Algorithmics. Theory & Practice*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[6] R. BAEZA-YATES, G. H. GONNET, M. REGNIER, *Analysis of Boyer-Moore type string searching algorithms*, Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 1990, pp. 328–343.

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

[8] M. CROCHEMORE, A. CZUMAJ, L. GASINIEC, S. JAROMINEK, T. LECROQ, W. PLANDOWSKI, AND W. RYTTER, *Speeding up two string-matching algorithms*, Proceedings of the Ninth Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, Lecture Notes in Computer Science, 577, Springer Verlag, Berlin, New York, 1992, pp. 589–602.

[9] R. COLE AND R. HARIHARAN, *Tighter bounds on the exact complexity of string matching*, Proceedings of the Thirty Third Annual IEEE Symposium on the Foundations of Computer Science, Pittsburgh, 1992, pp. 600–609.

[10] R. COLE, R. HARIHARAN, M. PATERSON, AND U. ZWICK, *Which patterns are hard to find*, to appear, this journal.

[11] Z. GALIL AND R. GIANCARLO, *On the exact complexity of string matching: upper bounds*, SIAM J. Comput., 21 (1992), pp. 407–437.

[12] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, McGraw Hill, New York, 1990.

[13] M. CROCHEMORE AND D. PERRIN, *Two-way pattern matching*, Tech. report, Laboratoire Informatique, Théorique et Programmation, Université Paris 7, 1989.

[14] Z. GALIL, *On improving the worst case running time of the Boyer-Moore string matching algorithm*, Comm. ACM, 22 (1979), pp. 505–508.

[15] ———, *String matching in real time*, J. ACM, 1 (1981), pp. 134–149.

[16] Z. GALIL AND K. PARK, *An improved algorithm for approximate string matching*, Proceedings of the Sixteenth International Colloquium on Automata, Languages and Programming, Stresa, Italy, Lecture Notes in Comput. Sci. 372, Springer-Verlag, Berlin, New York, 1989, pp. 394–404.

[17] Z. GALIL AND J. SEIFERAS, *Time space optimal string matching*, J. Comput. System. Sci., 26 (1983), pp. 280–294.

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

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

[20] R. KARP AND M. O. RABIN, *Efficient randomized pattern matching algorithms*, IBM J. Res. Develop., 2 (1987), pp. 249–260.

[21] G. M. LANDAU AND U. VISHKIN, *Fast parallel and serial approximate string matching*, J. Algorithms, 10 (1989), pp. 262–272.

[22] M. LOTHAIRE, *Combinatorics on Words*, Addison-Wesley, Reading, MA, 1982.

[23] W. RYTTER, *A correct preprocessing algorithm for Boyer-Moore string searching*, SIAM J. Comput., 9 (1980), pp. 509–512.

[24] D. D. SLEATOR AND R. E. TARJAN, *Amortized efficiency of list update rules*, Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, Washington D.C., 1984, pp. 488–492.

[25] E. UKKONEN, *Finding approximate patterns in strings*, J. Algorithms, 6 (1985), pp. 132–137.

[26] U. VISHKIN, *Optimal pattern matching in strings*, Informa. Control, 67 (1985), pp. 91–113.

[27] U. VISHKIN, *Deterministic sampling — A new technique for fast pattern matching*, Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing, Baltimore, MD, 1990, pp. 170–180.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1093–1137, December 1994

001

A GRAMMAR-BASED APPROACH TOWARDS UNIFYING HIERARCHICAL DATA MODELS*

MARC GYSSENS†, JAN PAREDAENS‡, AND DIRK VAN GUCHT§

Abstract. A simple model for representing the hierarchical structure of information is proposed. This model, called the grammatical model, is based on trees that are generated by grammars; the grammars describe the hierarchy of the information represented by the trees. Two methods for querying in this data model are given. The first, called the grammatical algebra, is based on a set of primitive grammar-oriented operators, the second, called the grammatical calculus, on local transformations on the trees. The semantics of both is formally defined. Decidability issues regarding the grammatical calculus are investigated. Finally, the two querying methods are proved to be equally expressive.

Key words. information base, grammars, trees, transformations, algebra, calculus

AMS subject classification. 68P15

1. Introduction. Until the mid-1980s much attention was paid to the relational database model (see, e.g., [17], [18], and [21]). We were intrigued by its simplicity, both for modeling and manipulating data. Recently, however, we became aware of its drawbacks when trying to model data applications beyond the traditional business-oriented applications, such as CAD-CAM, office automation, and text-oriented and multimedia databases. Therefore, a great number of data models have been proposed as a possible successor of the relational model.

Semantic data models, such as ER [7], FDM [19], SDM [12], Format [13], and IFO [2], provide a rich set of design tools for representing the complex interrelationships of data. These tools are typically variants of familiar aggregation, generalization, and set-formation constructs. Although query languages have been defined for some semantic data models, their main purpose is to provide database design tools that are more powerful than the modeling tools of the relational model. The logic-based models, such as Datalog [21], LDM [16], and LDL [4], zero in on the limited expressiveness of data manipulation languages of the relational model, i.e., they generalize the relational calculus to express queries that can be specified recursively. Finally, there are the relational extensions of the standard relational model, such as RM/T [8] and the nested relational model [9], [14], [20]. These models try to strike a balance between the elegance of the relational model and the expressiveness of semantic data models. In other words, they are not as rich as the semantic models in their modeling power, but they provide simple yet powerful extensions of the relational model.

Although there exist significant differences between all these models, they share the property that they recognize as the most fundamental characteristic of data its hierarchical structure. On the other hand, however, it is not quite clear whether they can effectively model all data applications which exhibit a hierarchical nature. A good example are textbases [11], which in addition to having a hierarchical structure, are constructed out of rules that follow a grammatical structure. Grammatical structures are also implicitly present in, e.g., VERSO [1], [5], a variation of the nested relational model, where at some level data are structured as regular expressions.

It is the intention of this paper to use a simple and a well-known model as a unifying skeleton to describe the hierarchy in an information base as well as the grammatical structure

*Received by the editors September 26, 1990; accepted for publication (in revised form) June 29, 1993.

†Department WNI, University of Limburg (LUC), B-3590 Diepenbeek, Belgium (gyssens@charlie.luc.ac.be).

‡Department of Math and Computer Science, University of Antwerp (UIA), B-2610 Antwerpen, Belgium.

§Computer Science Department, Indiana University, Bloomington, Indiana 47405-4101.

of texts. This model has been presented informally in [11]. It is called the grammatical model and is based on grammars introduced some 30 years ago to study the syntax of programming and formal languages (see, e.g., [10]). The grammatical model will benefit greatly from the clear understanding of the grammars and from their major importance for computer science.

It seems natural to formalize a hierarchical structure by a tree. Therefore, we represent information as a tree which can be generated by a grammar. Each leaf of the tree represents an object, and the internal nodes represent the relationship between the objects. The grammar specifies the scheme, i.e., the overall structure between these relationships. In this way, the information about two employees (one manager and one worker) will be represented by the tree in Fig. 1.

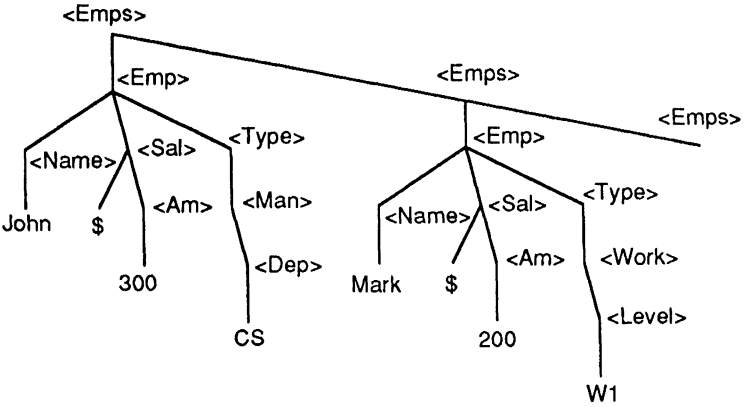


FIG. 1. *Tree representing information about employees.*

This tree is generated by a grammar with the following productions:

P1 : $\langle Emps\rangle \rightarrow \langle Emp\rangle\langle Emps\rangle$ P4 : $\langle Type\rangle \rightarrow \langle Work\rangle$ P6 : $\langle Work\rangle \rightarrow \langle Level\rangle$
P2 : $\langle Emp\rangle \rightarrow \langle Name\rangle\langle Sal\rangle\langle Type\rangle$ P5 : $\langle Man\rangle \rightarrow \langle Dep\rangle$ P7 : $\langle Sal\rangle \rightarrow \$\langle Am\rangle$
P3 : $\langle Type\rangle \rightarrow \langle Man\rangle$.

Notice that production P1 specifies information as a list, which is used here to model a set. Production P2 specifies an employee as the aggregation of his/her name, salary, and type. Productions P3 and P4 define the type of an employee as either manager or worker, which is an example of generalization, or, alternatively viewed, an example of specialization. This example illustrates that the grammatical model allows for the three basic constructs of most data models (set-formation, generalization, and aggregation) in a uniform way. Finally notice production P7. The grammatical model allows, in a convenient way, for additional syntactic features; e.g., the terminal symbol "$" indicates that the salary is expressed in dollar amounts.

For hierarchically organized data, the tree structure of the grammatical model also has some advantages with respect to implementation. Contrary to the relational and the nested relational models, for example, the straightforward implementation of the grammatical model is feasible, yielding a physical representation that is fairly close to the conceptual representation.

Turning to the dynamical aspects of the grammatical model, we see that there are two obvious ways to express queries by transforming the trees. A first method consists of defining operators that locally transform the trees and the grammars. A second way consists of describing this transformation in a less procedural way, indicating the relationship between the given trees and the result trees. For reasons that are evident these methods are called the algebra and the calculus, respectively. They are proved to be equivalent. Although the calculus is the more natural of both methods, its formal semantics need a detailed description to handle all possible problems of cyclicity and ambiguity.

The paper is organized as follows. In §2, we give the basic definitions of the grammatical model. In §3, we define an algebra to query and manipulate information bases defined over the grammatical model. Section 4 introduces an alternative query mechanism called the grammatical calculus. The grammatical calculus is based on pattern-matching. Section 5 briefly discusses some decidability issues regarding the grammatical calculus. In §6, we show that the grammatical algebra and calculus are equivalent with respect to expressive power. Finally, §7 proposes some directions for future research.

2. The grammatical model. Throughout this paper, we assume the reader is familiar with the basic terminology concerning trees (e.g., [3]) and formal languages (e.g., [10]). As stated in §1, we shall represent an information base as a tree, the structure of which is controlled by a formal grammar. We shall borrow the terms "scheme" and "instance" from the relational model and use the former to indicate the grammar and the latter to indicate the tree.

DEFINITION 2.1. An *information base scheme* is a grammar $\mathcal{G} = (V, T, S, P)$ with

- V being a finite set of attributes;
- T being a finite set of constants;
- S being a set of axioms, $S \subseteq V$;
- P being a finite set of productions of the form $A \rightarrow s$ where
  - –$A \in V$,
  - –$s \in (V \cup T)^*$,
  - –each attribute appears at most once in s.[1]

Actual data will be represented in a tree the internal nodes of which are labeled by attributes. Note that we do not require the leaves to be labeled by constants; if a leaf is labeled by an attribute, this simply means there are no data known for that attribute.

DEFINITION 2.2. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. A *data-tree* (D-*tree*) over $\mathcal{G}$ is a tree whose nodes are labeled with elements of $V \cup T$ in such a way that each internal node is labeled by an attribute. The set of all D-trees over $\mathcal{G}$ is denoted $\mathcal{D}(\mathcal{G})$. The empty D-tree is denoted **E**.

In the upcoming sections, we shall frequently use some operations on D-trees, which will be denoted as in Definition 2.3.

DEFINITION 2.3. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let **D** be a D-tree over $\mathcal{G}$. Then

- if **n** is a node of **D**, lbl(**n**) denotes its label;
- **rt(D)** denotes the root of **D** and rt(**D**) the label of the root of **D**;
- if **n** is a node of **D** with **n**$\neq$ **rt(D)**, **par(n)** denotes its parent and par(**n**) the label of its parent;
- if **n** is a node of **D**, **chln(n)** is the sequence of all its children, **chtrs(n)** is the sequence of the subtrees of **D** whose roots are the children of **n**, and chln(**n**) is the sequence of the labels of all the children of **n**.

Finally, a D-tree over an information base scheme that is also a derivation tree will be called an information base instance.

DEFINITION 2.4. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. A D-tree **D** over $\mathcal{G}$ is called an *information base instance* over $\mathcal{G}$ if

- rt(**D**) $\in S$;
- for each internal node **n** in **D**, the production lbl(**n**) $\rightarrow$ chln(**n**) is in P.

[1] The empty string is denoted ε.

Example 2.5. Consider the information base scheme $\mathcal{G} = (V, T, S, P)$ with

- $V = \{Fams, Fam, Father, Mother, Children, Child, String, Chr\}$;
- $T = \{A, \ldots, Z, a, \ldots, z\}$;
- $S = \{Families\}$;
- $$P = \{\begin{array}{ll} \langle Fams\rangle \longrightarrow \langle Fam\rangle\langle Fams\rangle & \langle Child\rangle \longrightarrow \langle String\rangle \\ \langle Fam\rangle \longrightarrow \langle Father\rangle\langle Mother\rangle\langle Children\rangle & \langle String\rangle \longrightarrow \langle Chr\rangle\langle String\rangle \\ \langle Children\rangle \longrightarrow \langle Child\rangle\langle Children\rangle & \langle Chr\rangle \longrightarrow A \\ \langle Father\rangle \longrightarrow \langle String\rangle & \quad\vdots \\ \langle Mother\rangle \longrightarrow \langle String\rangle & \langle Chr\rangle \longrightarrow z\} \end{array}$$

representing the structure on information base concerning families with their children. Note that this information base scheme also includes an "implementation" of strings. For example, the string *Ian* must be represented as the D-tree in Fig. 2.

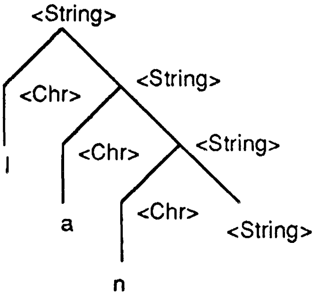


FIG. 2. *Representation of a string.*

However, we might have considered strings as elements of some set, sufficiently large for our purposes, rather than as a sequence of characters. In the upcoming sections we shall not bother with this low-level representation, since this is not our main concern. From now on, we shall no longer write productions for attributes such as $\langle String\rangle$. With this in mind, the D-tree in Fig. 3 represents an information base instance over $\mathcal{G}$ showing two families, the former composed of *Ian* and *Mary* with children *Brian* and *Wendy* and the latter composed of *Nick* and *Brenda* with no children.

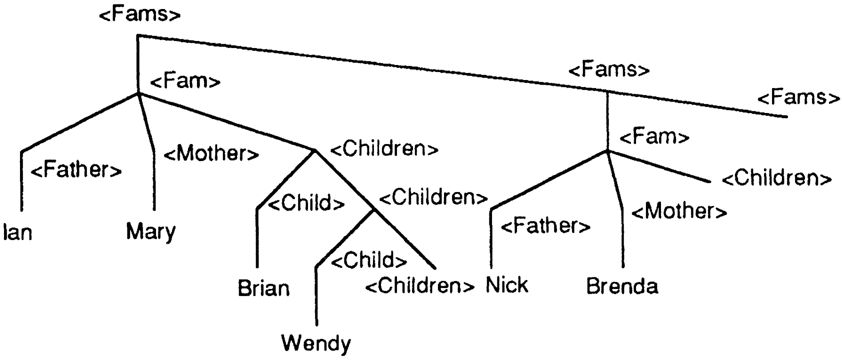


FIG. 3. *An instance of an information base about families.*

Grammar-based models turn out to be highly appropriate for representing text-dominated databases, as was observed by Gonnet and Tompa. The following example is inspired by [11].

Example 2.6. Consider an information base scheme with the following productions:

$\langle \mathit{Refs} \rangle \longrightarrow \langle \mathit{Ref} \rangle \langle \mathit{Refs} \rangle$ $\qquad \langle \mathit{Source} \rangle \longrightarrow \langle \mathit{Journ} \rangle \langle \mathit{Issue} \rangle$
$\langle \mathit{Ref} \rangle \longrightarrow \langle \mathit{Auths} \rangle \langle \mathit{Tit} \rangle \langle \mathit{Source} \rangle \langle \mathit{Year} \rangle$ $\qquad \langle \mathit{Source} \rangle \longrightarrow \langle \mathit{Book} \rangle$
$\langle \mathit{Auths} \rangle \longrightarrow \langle \mathit{Auth} \rangle \langle \mathit{Auths} \rangle$ $\qquad \langle \mathit{Issue} \rangle \longrightarrow \langle \mathit{Vol} \rangle : \langle \mathit{Nr} \rangle.$

The attributes not mentioned in the left-hand side of some production are supposed to take either a string or a number as a value. Part of some information base instance is represented in Fig. 4.

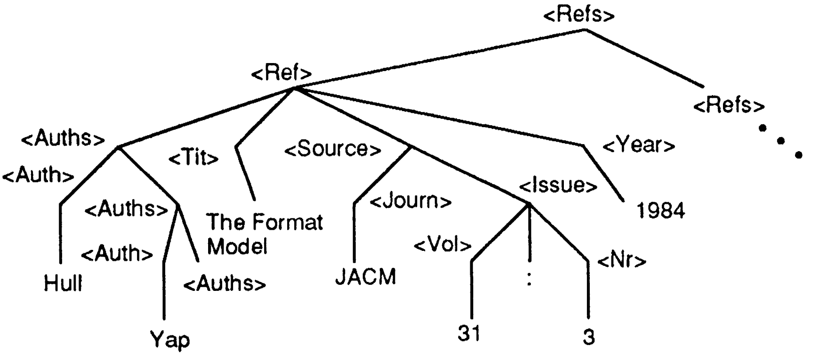


FIG. 4. *An instance of a bibliographic information base.*

We will also need the notion of isomorphic D-trees:

DEFINITION 2.7. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}_1$ and $\mathbf{D}_2$ be D-trees over $\mathcal{G}$. $\mathbf{D}_1$ and $\mathbf{D}_2$ are said to be *isomorphic*, denoted $\mathbf{D}_1 \cong \mathbf{D}_2$, if there exists a mapping between the nodes of $\mathbf{D}_1$ and $\mathbf{D}_2$ that is one to one and onto, preserving the labels and the tree structure. Isomorphism is extended to finite sequences of D-trees in the canonical way.

3. An algebra for transforming information bases. In this section, we propose an algebraic language for the manipulation of grammatically defined information bases that not only allows us to formulate queries, but also to apply more general transformations. Each operator is defined both on scheme and on instance level. Here we implicitly assume that only one information base instance is considered at a time.

The algebra we propose consists of eight basic operators, defined below. At the same time, we also define some derived operators, both as an illustration and because we need them further on.

First, we define three types of substitutions, which do not alter the structure of an information base instance, but only change (attribute) labels.

The parent substitution $\Sigma\pi[A \to s, B]$ substitutes by B all attributes A from which s is derived.

DEFINITION 3.1. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s \in P$, or $A \in V$ and $s = \varepsilon$. Let B be an attribute (B does not have to be in V), and suppose that A and B never occur simultaneously in the right-hand side of a production of P. The *parent substitution* is defined as follows:

- $\Sigma\pi[A \to s, B](\mathcal{G}) = \mathcal{G}' = (V', T, S', P')$ where
  - $V' = V \cup \{B\}$;
  - if $A \in S$, then $S' = S \cup \{B\}$, else $S' = S$;
  - Let $P'' = (P - \{A \to s\}) \cup \{B \to s\}$. Then

$$P' = P'' \cup \{C \to s_1 B s_2 \mid C \in V',\ s_1 s_2 \in (V \cup T)^*,\ \text{and}\ C \to s_1 A s_2 \in P''\}.$$

- $\Sigma\pi[A \to s, B](\mathbf{D})$ is obtained by simultaneously relabeling by B each node $\mathbf{n}$ in $\mathbf{D}$ with $\mathrm{lbl}(\mathbf{n}) = A$ and $\mathrm{chln}(\mathbf{n}) = s$.

Note that the condition of A and B not occurring simultaneously in the right-hand side of a production of $\mathcal{G}$ prevents B from appearing more than once in the right-hand side of a production of $\mathcal{G}'$.

The child substitution $\Sigma\chi[A \to s_1Bs_2, B, C]$ substitutes by C all attributes B in a string s_1Bs_2 that is derived from A.

DEFINITION 3.2. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $B \in V$, and let $A \to s_1Bs_2 \in P$. Let C be an attribute (C does not have to be in V), and suppose that C does not occur in s_1s_2. The *child substitution* is defined as follows:

- $\Sigma\chi[A \to s_1Bs_2, B, C](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{C\}$;
  - Let $P'' = (P - \{A \to s_1Bs_2\}) \cup \{A \to s_1Cs_2\}$. Then

$$P' = P'' \cup \{C \to s \mid s \in (V' \cup T)^* \text{ and } B \to s \in P''\}.$$

- $\Sigma\chi[A \to s_1Bs_2, B, C](\mathbf{D})$ is obtained by simultaneously relabeling by C each internal node $\mathbf{n}$ in $\mathbf{D}$ with $\mathrm{lbl}(\mathbf{n}) = B$, $\mathrm{par}(\mathbf{n}) = A$ and $\mathrm{chln}(\mathbf{par}(\mathbf{n})) = s_1Bs_2$.

As before, the condition on C prevents illegal substitutions.

Finally, we define child equality substitution. The child equality substitution $\Sigma\epsilon[A \to s_1Bs_2, B \to s_3, C, D]$ substitutes by D all attributes B in a string s_1Bs_2 that is derived from A and from which s_3 is derived, provided B has both a sibling and a child labeled C that define isomorphic subtrees.

DEFINITION 3.3. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1Bs_2, B \to s_3 \in P$. Let $C \in V$, and suppose that C occurs both in s_1s_2 and s_3. Let D be an attribute (D does not have to be in V), and suppose that D does not occur in s_1s_2. The *child equality substitution* is defined as follows:

- $\Sigma\epsilon[A \to s_1Bs_2, B \to s_3, C, D](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{D\}$;
  - if $A = B$, then $P' = P \cup \{A \to s_1Ds_2, D \to s_3, D \to s_1Ds_2\}$, else $P' = P \cup \{A \to s_1Ds_2, D \to s_3\}$.
- Let $\mathbf{n}$ be an internal node in $\mathbf{D}$ with $\mathrm{lbl}(\mathbf{n}) = B$, $\mathrm{par}(\mathbf{n}) = A$, $\mathrm{chln}(\mathbf{par}(\mathbf{n})) = s_1Bs_2$, and $\mathrm{chln}(\mathbf{n}) = s_3$. Let $\mathbf{m}_1$ be the sibling and $\mathbf{m}_2$ be the child of $\mathbf{n}$ with label C and let $\mathbf{D}_1$ and $\mathbf{D}_2$ be the subtrees of $\mathbf{D}$ with $\mathbf{rt}(\mathbf{D}_1) = \mathbf{m}_1$ and $\mathbf{rt}(\mathbf{D}_2) = \mathbf{m}_2$. Then $\Sigma\epsilon[A \to s_1Bs_2, B \to s_3, C, D](\mathbf{D})$ is obtained by simultaneously relabeling by D each such node $\mathbf{n}$ for which $\mathbf{D}_1 \cong \mathbf{D}_2$.

Example 3.4. Consider an information base scheme $\mathcal{G} = (V, T, S, P)$ with $V = \{A, B, C, D\}$, $T = \{a, b\}$, $S = \{A\}$, and

$$P = \{A \to BA, A \to B, B \to CD, C \to a, C \to b, D \to a, D \to b\},$$

and let $\mathbf{D}$ be the information base instance over $\mathcal{G}$ shown in Fig. 5.

Then the parent substitution $\Sigma\pi[A \to BA, E]$ yields the information base scheme $\mathcal{G}' = (V', T, S', P')$ with $V' = \{A, B, C, D, E\}$, $S' = \{A, E\}$, and

$$P' = \{E \to BA, A \to B, B \to CD, C \to a, C \to b, D \to a, D \to b, E \to BE\}$$

and the information base instance $\mathbf{D}'$ in Fig. 6.

The child substitution $\Sigma\chi[E \to BA, A, E]$ applied to the information base thus obtained yields the information base scheme $\mathcal{G}'' = (V', T, S', P'')$ with

$$P'' = \{A \to B, B \to CD, C \to a, C \to b, D \to a, D \to b, E \to BE, E \to B\}$$

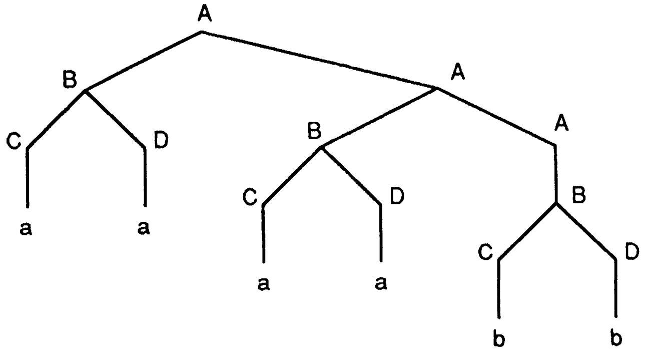


FIG. 5. *An information base instance* **D**.

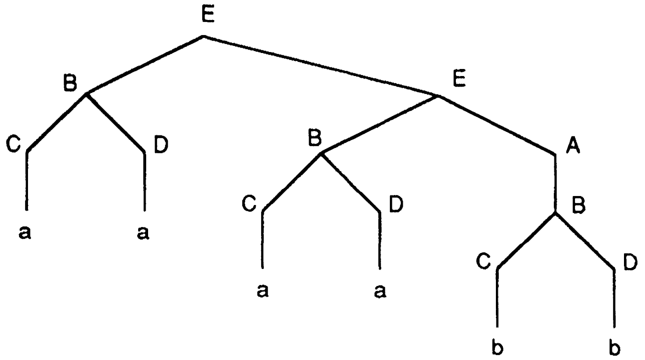


FIG. 6. *The information base instance* $\mathbf{D}' = \Sigma\pi[A \rightarrow BA, E](\mathbf{D})$.

and the information base instance $\mathbf{D}''$ in Fig. 7.

Finally, the child equality substitution $\Sigma\epsilon[E \rightarrow BE, E \rightarrow BE, B, D]$ applied to the last result yields the information base scheme $\mathcal{G}''' = (V', T, S', P''')$ with

$$P''' = \{A \rightarrow B, B \rightarrow CD, C \rightarrow a, C \rightarrow b, D \rightarrow a, D \rightarrow b,$$
$$E \rightarrow BE, E \rightarrow B, E \rightarrow BD, D \rightarrow BE, D \rightarrow BD\}$$

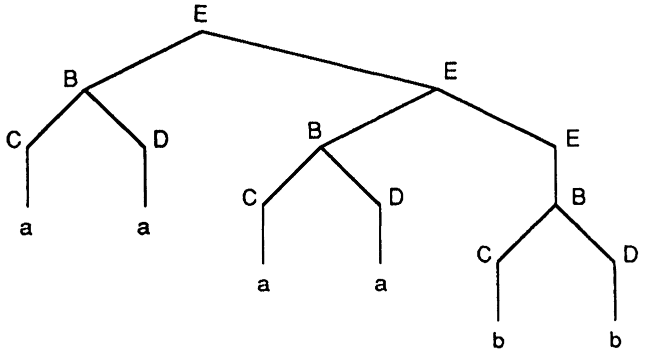


FIG. 7. *The information base instance* $\mathbf{D}'' = \Sigma\chi[E \rightarrow BA, A, E](\mathbf{D}')$.

and the information base instance $\mathbf{D}'''$ in Fig. 8.

Next, we define two operators that allow the introduction of new nodes and the removal of existing ones.

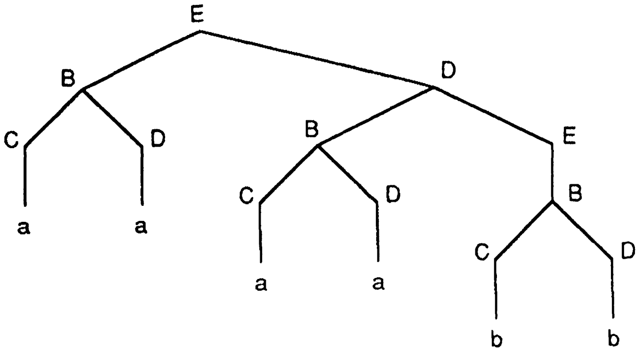


FIG. 8. *The information base instance* $\mathbf{D}''' = \Sigma\epsilon[E \rightarrow BE, E \rightarrow BE, B, D](\mathbf{D}'')$.

The node insertion $N\iota[A \rightarrow s_1s_2s_3, s_1Bs_3]$ inserts in each derivation of $s_1s_2s_3$ from A a node B as a child of A and the father of s_2.

DEFINITION 3.5. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \rightarrow s_1s_2s_3 \in P$, or $A \in V$ and $s_1s_2s_3 = \varepsilon$. Let B be an attribute not in V. The *node insertion* is defined as follows:

- $N\iota[A \rightarrow s_1s_2s_3, s_1Bs_3](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{B\}$;
  - $P' = (P - \{A \rightarrow s_1s_2s_3\}) \cup \{A \rightarrow s_1Bs_3, B \rightarrow s_2\}$.
- Let $\mathbf{n}$ be a node of $\mathbf{D}$ with $\text{lbl}(\mathbf{n}) = A$ and $\text{chln}(\mathbf{n}) = s_1s_2s_3$. Then $N\iota[A \rightarrow s_1s_2s_3, s_1Bs_3](\mathbf{D})$ is obtained by simultaneously inserting for each such node $\mathbf{n}$ a node $\mathbf{n}'$ for which $\text{lbl}(\mathbf{n}') = B$, $\mathbf{par}(\mathbf{n}') = \mathbf{n}$, and $\mathbf{chln}(\mathbf{n}')$ is the subinterval of $\mathbf{chln}(\mathbf{n})$ corresponding to s_2.

The node deletion $N\delta[A]$ deletes each subtree whose root is labeled by A.

DEFINITION 3.6. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \in V$. The *node deletion* is defined as follows:

- $N\delta[A](\mathcal{G}) = \mathcal{G}' = (V', T, S', P')$ where
  - $V' = V - \{A\}$;
  - $S' = S - \{A\}$;
  - Let $P'' = \{B \rightarrow s \mid B \in V',\ s \in (V' \cup T)^*, \text{ and } B \rightarrow s \in P\}$. Then

$$P' = P'' \cup \{B \rightarrow s_1s_2 \mid B \in V',\ s_1s_2 \in (V' \cup T)^*, \text{ and } B \rightarrow s_1As_2 \in P\}.$$

- $N\delta[A](\mathbf{D})$ is obtained by deleting each subtree $\mathbf{D}'$ from $\mathbf{D}$ with $\text{rt}(\mathbf{D}') = A$.

Example 3.7. Consider an information base scheme $\mathcal{G} = (V, T, S, P)$ with $V = \{A, B, C, D, E\}$, $T = \{a, b, c, d, e\}$, $S = \{A\}$, and

$$P = \{A \rightarrow BCD, B \rightarrow abEd, C \rightarrow c, D \rightarrow Bd, E \rightarrow e\},$$

and let $\mathbf{D}$ be the information base instance over $\mathcal{G}$ shown in Fig. 9.

Then the node insertion $N\iota[B \rightarrow abEd, aFd]$ yields the information base scheme $\mathcal{G}' = (V', T, S, P')$ with $V' = \{A, B, C, D, E, F\}$ and

$$P' = \{A \rightarrow BCD, B \rightarrow aFd, C \rightarrow c, D \rightarrow Bd, E \rightarrow e, F \rightarrow bE\}$$

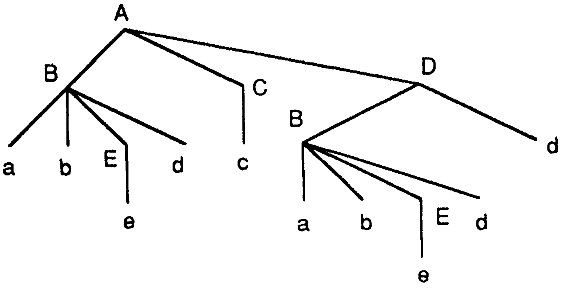


FIG. 9. *An information base instance* **D**.

and the information base instance **D**$'$ in Fig. 10.

The node deletion N$\delta[F]$ applied to the information base thus obtained yields the information base scheme $\mathcal{G}'' = (V'', T, S, P'')$ with $V'' = \{A, B, C, D, E\}$ and

$$P'' = \{A \rightarrow BCD, B \rightarrow ad, C \rightarrow c, D \rightarrow Bd, E \rightarrow e\}$$

and the information base instance **D**$''$ of Fig. 11.

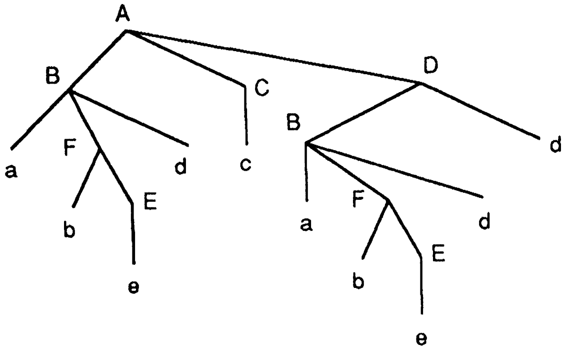


FIG. 10. *The information base instance* $\mathbf{D}' = \mathrm{N}\iota[B \rightarrow abEd, aFd](\mathbf{D})$.

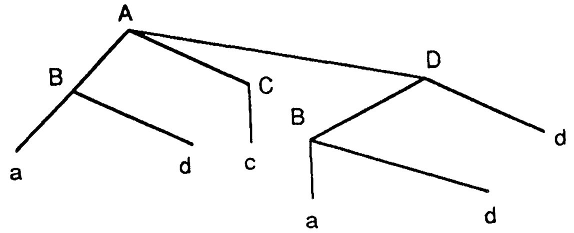


FIG. 11. *The information base instance* $\mathbf{D}'' = \mathrm{N}\delta[F](\mathbf{D})$.

Note that N$\delta[A]$ applied to any of the above information bases would yield the empty information base.

We also need two operators that copy information from one place in an information base to another. They are defined recursively.

Essentially, the downward duplication $\Delta\delta[A \rightarrow s_1Bs_2, B \rightarrow s_3, C, D]$ copies the subtree with root C in a string s_1Bs_2 derived from A as the rightmost sibling of a string s_3 which is derived from B, and renames the root of that copy to D. What makes the definition below somewhat involved is that, for reasons of uniformity, we have to require that this duplication is propagated into these subtrees as well.

DEFINITION 3.8. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1Bs_2 \in P$, and let either $B \to s_3 \in P$ or $s_3 = \varepsilon$. Let $C \in V$, and suppose that C occurs in s_1s_2. Let D be an attribute (D does not have to be in V), and suppose that D does not occur in s_3. The *downward duplication* is recursively defined as follows:

- $\Delta\delta[A \to s_1Bs_2, B \to s_3, C, D](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{D\}$;
  - $P' = P \cup \{B \to s_3D\} \cup \{D \to s \mid s \in (V \cup T)^* \text{ and } C \to s \in P\}$.
- Let $\mathbf{n}$ be a internal node in $\mathbf{D}$ with $\text{lbl}(\mathbf{n}) = B$, $\text{par}(\mathbf{n}) = A$, $\text{chln}(\mathbf{par}(\mathbf{n})) = s_1Bs_2$, and $\text{chln}(\mathbf{n}) = s_3$. Let $\mathbf{m}$ be the node in $\mathbf{chln}(\mathbf{par}(\mathbf{n}))$ with $\text{lbl}(\mathbf{m}) = C$. Let $\mathbf{D}''$ be the subtree of $\mathbf{D}$ defined by $\mathbf{rt}(\mathbf{D}'') = \mathbf{m}$, and let $\mathbf{D}''' = \Delta\delta[A \to s_1Bs_2, B \to s_3, C, D](\mathbf{D}'')$. Then $\Delta\delta[A \to s_1Bs_2, B \to s_3, C, D](\mathbf{D})$ is obtained by simultaneously adding to $\mathbf{chln}(\mathbf{n})$ for each such node $\mathbf{n}$ a rightmost sibling $\mathbf{n}'$ with $\text{lbl}(\mathbf{n}') = D$. The subtree $\mathbf{D}'$ of $\Delta\delta[A \to s_1Bs_2, B \to s_3, C, D](\mathbf{D})$ defined by $\mathbf{rt}(\mathbf{D}') = \mathbf{n}'$ is then determined by $\mathbf{chtrs}(\mathbf{n}') \cong \mathbf{chtrs}(\mathbf{rt}(\mathbf{D}'''))$.

Example 3.9. Consider an information base scheme $\mathcal{G} = (V, T, S, P)$ with $V = \{A, B, C\}$, $T = \{a, b, c, d\}$, $S = \{A\}$, and $P = \{A \to BaC, B \to bd, C \to cA, C \to c\}$, and let $\mathbf{D}$ be the information base instance over $\mathcal{G}$ in Fig. 12.

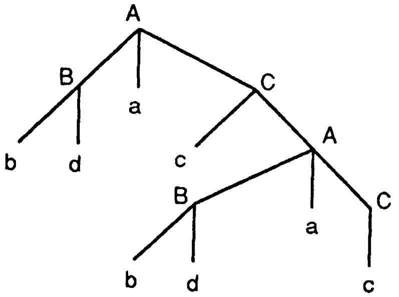


FIG. 12. *An information base instance* $\mathbf{D}$.

Then the downward duplication $\Delta\delta[A \to BaC, B \to bd, C, D]$ yields the information base scheme $\mathcal{G}' = (V', T, S, P')$ with $V' = \{A, B, C, D\}$ and

$$P' = \{A \to BaC, B \to bdD, C \to cA, C \to c, D \to cA, D \to c\}$$

and the information base instance $\mathbf{D}'$ of Fig. 13.

The downward duplication copies information downward into the tree; the upward duplication is its upward counterpart.

DEFINITION 3.10. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1Bs_2, B \to s_3 \in P$. Let $C \in V$, and suppose that C occurs in s_3. Let D be an attribute (D does not have to be in V), and suppose that D does not occur in s_1s_2. The *upward duplication* is recursively defined as follows:

- $\Delta\upsilon[A \to s_1Bs_2, B \to s_3, C, D](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{D\}$;
  - $P' = P \cup \{A \to s_1Bs_2D\} \cup \{D \to s \mid s \in (V \cup T)^* \text{ and } C \to s \in P\}$.
- Let $\mathbf{n}$ be a internal node in $\mathbf{D}$ with $\text{lbl}(\mathbf{n}) = B$, $\text{par}(\mathbf{n}) = A$, $\text{chln}(\mathbf{par}(\mathbf{n})) = s_1Bs_2$, and $\text{chln}(\mathbf{n}) = s_3$. Let $\mathbf{m}$ be the node in $\mathbf{chln}(\mathbf{n})$ with $\text{lbl}(\mathbf{m}) = C$. Let $\mathbf{D}''$ be the subtree of $\mathbf{D}$ defined by $\mathbf{rt}(\mathbf{D}'') = \mathbf{m}$, and let $\mathbf{D}''' = \Delta\upsilon[A \to s_1Bs_2, B \to s_3, C, D](\mathbf{D}'')$. Then $\Delta\upsilon[A \to$

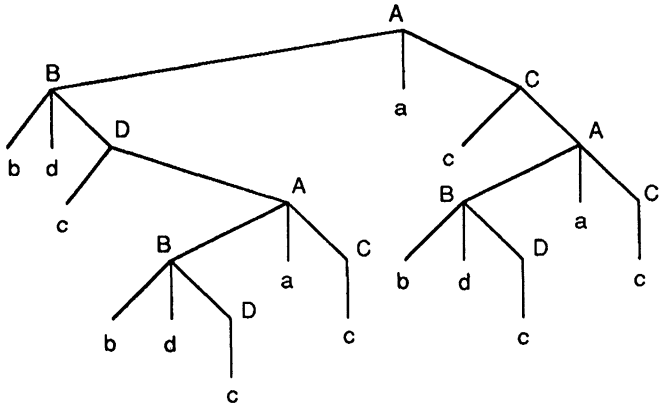


FIG. 13. *The information base instance* $\mathbf{D}' = \Delta\delta[A \rightarrow BaC, B \rightarrow bd, C, D](\mathbf{D})$.

$s_1Bs_2, B \rightarrow s_3, C, D](\mathbf{D})$ is obtained by simultaneously adding to **chln**(**par**(**n**)) for each such node **n** a rightmost sibling **n**′ with lbl(**n**′) $= D$. The subtree $\mathbf{D}'$ of $\Delta\upsilon[A \rightarrow s_1Bs_2, B \rightarrow s_3, C, D](\mathbf{D})$ defined by $\mathbf{rt}(\mathbf{D}') = \mathbf{n}'$ is then determined by **chtrs**(**n**′) $\cong$ **chtrs**(**rt**($\mathbf{D}'''$)).

Since upward duplication is very similar to downward duplication, we omit an example.

Using both downward and upward duplication, it is possible to simulate sidewise duplication, defined below.

DEFINITION 3.11. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let **D** be an information base instance over $\mathcal{G}$. Let $A \rightarrow s_1Cs_2 \in P$. Let D be an attribute (D does not have to be in V), and suppose that D does not occur in s_1s_2. The *sidewise duplication* is recursively defined as follows:

- $\Delta\sigma[A \rightarrow s_1Cs_2, C, D](\mathcal{G}) = \mathcal{G}' = (V', T, S, P')$ where
  - $V' = V \cup \{D\}$;
  - $P' = P \cup \{A \rightarrow s_1Cs_2D\} \cup \{D \rightarrow s \mid s \in (V \cup T)^*$ and $C \rightarrow s \in P\}$.

- Let **n** be a node in **D** with lbl(**n**) $= A$ and chln(**n**) $= s_1Cs_2$. Let **m** be the node in **chln**(**n**) with lbl(**m**) $= C$. Let $\mathbf{D}''$ be the subtree of **D** defined by $\mathbf{rt}(\mathbf{D}'') = \mathbf{m}$, and let $\mathbf{D}''' = \Delta\sigma[A \rightarrow s_1Cs_2, C, D](\mathbf{D}'')$. Then $\Delta\sigma[A \rightarrow s_1Cs_2, C, D](\mathbf{D})$ is obtained by simultaneously adding to **chln**(**n**) for each such node **n** a rightmost sibling **n**′ with lbl(**n**′) $= D$. The subtree $\mathbf{D}'$ of $\Delta\sigma[A \rightarrow s_1Cs_2, C, D](\mathbf{D})$ defined by $\mathbf{rt}(\mathbf{D}') = \mathbf{n}'$ is then determined by **chtrs**(**n**′) $\cong$ **chtrs**(**rt**($\mathbf{D}'''$)).

THEOREM 3.12. *Sidewise duplication can be expressed in terms of node insertion, downward duplication, upward duplication, and node deletion.*

Proof. Let $\mathcal{G} = (V, T, S, P)$ be the scheme of some information base, and consider the sidewise duplication $\Delta\sigma[A \rightarrow s_1Cs_2, C, D]$ in which A, C, D, s_1, and s_2 are as in Definition 3.11. Let E be an attribute not in V. Then $\Delta\sigma[A \rightarrow s_1Cs_2, C, D]$ can be performed by consecutively executing the following operations:

1. the node insertion $\mathrm{N}\iota[A \rightarrow s_1Cs_2, s_1Cs_2E]$;
2. the downward duplication $\Delta\delta[A \rightarrow s_1Cs_2E, E \rightarrow \varepsilon, C, C]$;
3. the upward duplication $\Delta\upsilon[A \rightarrow s_1Cs_2E, E \rightarrow C, C, D]$;
4. the node deletion $\mathrm{N}\Delta[E]$.

In step 1, the node E is actually created so that downward duplication can always be applied, even if $s_1s_2 = \varepsilon$. □

Note that the effects of downward duplication, upward duplication, and sidewise duplication can be easily undone by deleting D (using the notations used in the respective definitions). This observation yields a natural "embedding" of the nodes of the original information base instance into the resulting instance. From the definitions of the above operations, the result below follows in a straightforward manner.

THEOREM 3.13. *Let* $\mathbf{D}$ *be an information base instance over some appropriate scheme, and let* $\mathbf{D}'$ *be the resulting instance after a downward duplication, an upward duplication, or a sidewise duplication. Let* $\mathbf{n}_1$ *and* $\mathbf{n}_2$ *be two nodes in* $\mathbf{D}$, *and let* $\mathbf{n}'_1$ *and* $\mathbf{n}'_2$, *respectively, be the corresponding nodes in* $\mathbf{D}'$. *Let* $\mathbf{D}_1$ *and* $\mathbf{D}_2$ *be the subtrees of* $\mathbf{D}$ *defined by* $\mathbf{rt}(\mathbf{D}_1) = \mathbf{n}_1$ *and* $\mathbf{rt}(\mathbf{D}_2) = \mathbf{n}_2$ *and let* $\mathbf{D}'_1$ *and* $\mathbf{D}'_2$ *be the subtrees of* $\mathbf{D}'$ *defined by* $\mathbf{rt}(\mathbf{D}'_1) = \mathbf{n}'_1$ *and* $\mathbf{rt}(\mathbf{D}'_2) = \mathbf{n}'_2$. *Then* $\mathbf{D}'_1 \cong \mathbf{D}'_2$ *if and only if* $\mathbf{D}_1 \cong \mathbf{D}_2$. *Suppose furthermore that in* $\mathbf{D}'$ *the subtree* $\mathbf{D}'_3$ *is a 'duplicate" of* $\mathbf{D}_2$. *Let* $\mathbf{D}''_3$ *be any* D-*tree defined by* $\mathrm{rt}(\mathbf{D}''_3) = \mathrm{rt}(\mathbf{D}'_2)$ *and* $\mathbf{chtrs}(\mathbf{rt}(\mathbf{D}''_3)) = \mathbf{chtrs}(\mathbf{rt}(\mathbf{D}'_3))$. *Then* $\mathbf{D}'_2 \cong \mathbf{D}''_3$ *whence* $\mathbf{D}'_1 \cong \mathbf{D}''_3$ *if and only if* $\mathbf{D}_1 \cong \mathbf{D}_2$.

Theorem 3.13 will turn out to be essential in the proof of our main theorem, given in §6.

Finally, we introduce a permutation. Basically, a permutation recursively rearranges the children derived by some production $A \to s$.

DEFINITION 3.14. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1 \in P$, or $A \in V$ and $s_1 = \varepsilon$, and let $s_2 \in (V \cup T)^*$ contain the same attributes as s_1. The *permutation* is recursively defined as follows:

- $\Pi[A \to s_1, s_2](\mathcal{G}) = \mathcal{G}' = (V, T, S, P')$ where

$$P' = (P - \{A \to s_1\}) \cup \{A \to s_2\}.$$

- Let $\mathbf{n}$ be a node in $\mathbf{D}$ with $\mathrm{lbl}(\mathbf{n}) = A$ and $\mathrm{chln}(\mathbf{n}) = s_1$. $\Pi[A \to s_1, s_2](\mathbf{D})$ is obtained by first simultaneously substituting new nodes for $\mathbf{chln}(\mathbf{n})$ such that $\mathrm{chln}(\mathbf{n})$ becomes s_2. Now let B be an attribute in s_2, and let $\mathbf{m}$ be the node in $\Pi[A \to s_1, s_2](\mathbf{D})$ with $\mathrm{lbl}(\mathbf{m}) = B$ and $\mathbf{par}(\mathbf{m}) = \mathbf{n}$. Then the subtree $\mathbf{D}'$ of $\Pi[A \to s_1, s_2](\mathbf{D})$ defined by $\mathbf{rt}(\mathbf{D}') = \mathbf{m}$ is isomorphic to $\Pi[A \to s_1, s_2](\mathbf{D}'')$, where the D-tree $\mathbf{D}''$ is the subtree of $\mathbf{D}$ defined by $\mathrm{rt}(\mathbf{D}'') = B$ and $\mathbf{par}(\mathbf{rt}(\mathbf{D}'')) = \mathbf{n}$.

Our notion of permutation is somewhat wider than what is usually understood by this term. A permutation does indeed permute attributes, but can also insert, delete, and rearrange constants.

Example 3.15. Consider an information base scheme $\mathcal{G} = (V, T, S, P)$ with $V = \{A, B\}$, $T = \{a, b, c\}$, $S = \{A\}$, and $P = \{A \to aB, B \to bBc\}$, and consider the information base instance over $\mathcal{G}$ shown in Fig. 14 left.

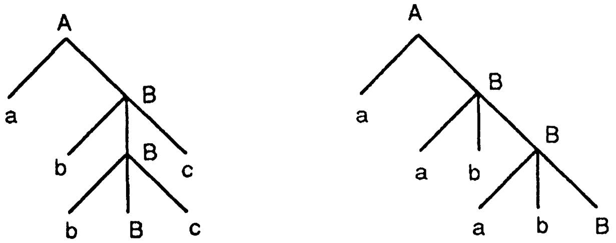


FIG. 14. *An example of a permutation.*

Then the permutation $\Pi[B \to bBc, abB]$ yields the information base scheme $\mathcal{G}' = (V, T, S, P')$ with $P' = \{A \to aB, B \to abB\}$ and the information base instance shown in Fig. 14 right.

Parent substitution, child substitution, child equality substitution, node insertion, node deletion, downward duplication, upward duplication, and permutation define the *grammatical algebra*. Many other conceivable operators can be expressed in terms of these eight, meaning there is a sequence of instance-independent grammatical algebra operations that returns the same result *at the instance level*. In general, however, it is unavoidable that the scheme returned by the algebra sequence defines a larger language than the scheme returned by the original operator (although in Theorem 3.12, they are equal).

We already saw that sidewise duplication can be expressed in the grammatical algebra. Below, we give two more examples of derived operations that are often needed in practical applications.

First, we introduce node merging. The node merging $\mathrm{N}\mu[A \to s_1Bs_2, B \to s_3]$ is obtained by pruning out each attribute B in a string s_1Bs_2 which is derived from A and from which s_3 is derived. In this way, $s_1s_3s_2$ will be derived from A instead.

DEFINITION 3.16. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1Bs_2 \in P$, and let $B \to s_3 \in P$, or $B \in V$ and $s_3 = \varepsilon$. Suppose that no attribute in s_3 appears in s_1s_2. The *node merging* is defined as follows:

- $\mathrm{N}\mu[A \to s_1Bs_2, B \to s_3](\mathcal{G}) = \mathcal{G}' = (V, T, S, P')$ where

$$P' = P \cup \{A \to s_1s_3s_2\}.$$

- Let $\mathbf{n}$ be a node of $\mathbf{D}$ with $\mathrm{lbl}(\mathbf{n}) = B$, $\mathrm{par}(\mathbf{n}) = A$, $\mathrm{chln}(\mathbf{par}(\mathbf{n})) = s_1Bs_2$, and $\mathrm{chln}(\mathbf{n}) = s_3$. Then $\mathrm{N}\mu[A \to s_1Bs_3, B \to s_3](\mathbf{D})$ is obtained by simultaneously substituting each such node $\mathbf{n}$ in $\mathbf{chln}(\mathbf{par}(\mathbf{n}))$ by $\mathbf{chln}(\mathbf{n})$.

Clearly, on instance level, a node insertion $\mathrm{N}\iota[A \to s_1s_2s_3, s_1Bs_3]$ can be undone by the node merging $\mathrm{N}\mu[A \to s_1Bs_3, B \to s_2]$. We now show the following.

THEOREM 3.17. *Node merging can be expressed in the grammatical algebra.*

Proof. Rather than giving a notationally cumbersome proof, we illustrate the general techniques that are needed on an example.

Consider an information base with scheme $\mathcal{G} = (V, T, S, P)$ where $V = \{A, B, C\}$, $T = \{a, b, c\}$, $S = \{A\}$, and $P = \{A \to aBc, B \to bC, C \to c\}$. The node merging $\mathrm{N}\mu[A \to aBc, B \to bC]$ can be expressed by consecutively performing the following operations:

1. the child substitution $\Sigma\chi[A \to aBc, B, B']$;
2. the upward duplication $\Delta\upsilon[A \to aB'c, B' \to bC, C, C]$;
3. the child substitution $\Sigma\chi[A \to aB'cC, B', B'']$;
4. the node deletion $\mathrm{N}\delta[B'']$;
5. the permutation $\Pi[A \to acC, abCc]$;
6. the child substitution $\Sigma\chi[A \to aB'c, B', B]$.

We invite the reader to check our claim on a concrete instance.

In general, step 2 must be carried out for each attribute in s_3. In step 5, all the copied attributes must be arranged in the right order and all the constants in s_3 inserted in the right place. □

Below, we give another example of a derived operator that will turn out to be very useful in the proof of our main result in §5.

DEFINITION 3.18. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $A \to s_1C_1s_2C_2s_3 \in P$ with $C_1 \neq A$ and $C_2 \neq A$. Let B be an attribute (B does not have to be in V), and suppose that A and B never occur simultaneously in the right-hand side of a production of P. The *parent equality substitution* is defined as follows:

- $\Sigma\sigma[A \to s_1C_1s_2C_2s_3, C_1, C_2, B](\mathcal{G}) = \mathcal{G}' = (V', T, S', P')$ where
  - $V' = V \cup \{B\}$;
  - if $A \in S$, then $S' = S \cup \{B\}$, else $S' = S$;

– Let $P'' = P \cup \{B \to s_1C_1s_2C_2s_3\}$. Then

$$P' = P'' \cup \{C \to s_4Bs_5 \mid C \in V',\ s_4s_5 \in (V \cup T)^*, \text{ and } C \to s_4As_5 \in P''\}.$$

• Let **n** be a node in **D** with lbl(**n**) $= A$ and **chln(n)** $= s_1C_1s_2C_2s_3$. Let $\mathbf{m}_1$ be the child of **n** with label C_1 and $\mathbf{m}_2$ be the child of **n** with label C_2 and let $\mathbf{D}_1$ and $\mathbf{D}_2$ be the subtrees of **D** with $\mathbf{rt}(\mathbf{D}_1) = \mathbf{m}_1$ and $\mathbf{rt}(\mathbf{D}_2) = \mathbf{m}_2$. Let $\mathbf{D}_3$ be the D-tree defined by $\mathbf{rt}(\mathbf{D}_3) = \mathbf{m}_1$ and $\mathbf{chtrs}(\mathbf{D}_3) = \mathbf{chtrs}(\mathbf{D}_2)$. Then $\Sigma\sigma[A \to s_1C_1s_2C_2s_3, C_1, C_2, B](\mathbf{D})$ is obtained by simultaneously relabeling by B each such node **n** for which $\mathbf{D}_1 \cong \mathbf{D}_3$.

The reader may wonder why we have imposed the restriction $C_1 \neq A$ and $C_2 \neq A$. Indeed, without this restriction, parent equality substitution would still be well defined. In the upcoming sections, however, we only need the restricted parent equality substitution, and, although Theorem 3.19 below still holds for the unrestricted parent equality substitution, the proof would become very involved.

THEOREM 3.19. *Parent equality substitution can be expressed in the grammatical algebra.*

Proof. Let $\mathcal{G} = (V, T, S, P)$ be the scheme of some information base, and consider the parent equality substitution $\Sigma\sigma[A \to s_1C_1s_2C_2s_3, C_1, C_2, B]$ (cf. Definition 3.18). Let D and E be attributes not in V. Then $\Sigma\sigma[A \to s_1C_1s_2C_2s_3, C_1, C_2, B]$ can be performed by consecutively performing the following operations:

1. the node insertion $\mathrm{N}\iota[A \to s_1C_1s_2C_2s_3, s_1C_1s_2Ds_3]$;
2. the child substitution $\Sigma\chi[A \to s_1C_1s_2Ds_3, C_1, C_2]$;
3. the child equality substitution $\Sigma\epsilon[A \to s_1C_2s_2Ds_3, D \to C_2, C_2, E]$;
4. the parent substitution $\Sigma\pi[A \to s_1C_2s_2Es_3, B]$;
5. the child substitution $\Sigma\chi[B \to s_1C_2s_2Es_3, C_2, C_1]$;
6. the child substitution $\Sigma\chi[A \to s_1C_2s_2Ds_3, C_2, C_1]$;
7. the node merging $\mathrm{M}\mu[B \to s_1C_1s_2Es_3, E \to C_2]$;
8. the node merging $\mathrm{M}\mu[A \to s_1C_1s_2Ds_3, D \to C_2]$. □

We now return to the bibliographical Example 2.6 to illustrate on a more realistic information base how the grammatical algebra can be used to solve queries or to perform transformations.

Example 3.20. Reconsider the information base of Example 2.6. Suppose we want to extract only the information on journal titles (between double quotes) with the name of the journal and the volume. The instance of Fig. 4 would then be transformed into the instance of Fig. 15.

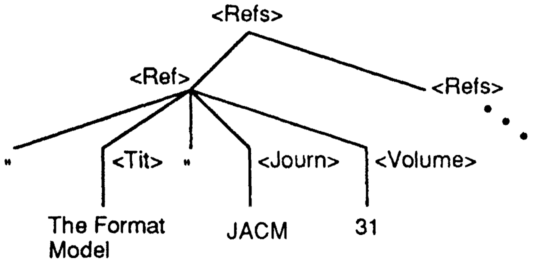


FIG. 15. *A transformation of the information base instance of Fig.* 4.

The transformation can be accomplished by consecutively performing the following operations:

1. $\mathrm{N}\delta[\langle Auths\rangle]$;
2. $\mathrm{N}\delta[\langle Year\rangle]$;
3. $\Delta\upsilon[\langle Source\rangle \to \langle Journ\rangle\langle Issue\rangle, \langle Issue\rangle \to \langle Vol\rangle : \langle Nr\rangle, \langle Vol\rangle, \langle Volume\rangle]$;

4. $\mathrm{N}\delta[\langle \mathit{Issue} \rangle]$;
5. $\mathrm{N}\mu[\langle \mathit{Ref} \rangle \rightarrow \langle \mathit{Tit} \rangle\langle \mathit{Source} \rangle, \langle \mathit{Source} \rangle \rightarrow \langle \mathit{Journ} \rangle\langle \mathit{Volume} \rangle]$;
6. $\Pi[\langle \mathit{Ref} \rangle \rightarrow \langle \mathit{Tit} \rangle\langle \mathit{Journ} \rangle\langle \mathit{Volume} \rangle,\text{''}\ \langle \mathit{Tit} \rangle\text{''}\langle \mathit{Journ} \rangle\langle \mathit{Volume} \rangle]$.

Observe that the instance obtained in the above example can be considered as a representation of a flat relational database model relation in the grammatical model. Below, we show how unary relational algebra operators can be performed. Note that, in order to simulate union, difference, and join, we need binary operators on information bases. This is beyond the scope of the present paper, however.

Example 3.21. Consider the following relational database relation R:

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
	$\vdots$	

This relation can be represented as an information base with scheme $\mathcal{G} = (V, T, S, P)$ where $V = \{U, R, A, B, C\}$, $T = \{a_1, \ldots, b_1, \ldots, c_1, \ldots\}$, $S = \{U\}$, and

$$P = \{U \rightarrow RU, R \rightarrow ABC, A \rightarrow a_1, \ldots, B \rightarrow b_1, \ldots, C \rightarrow c_1, \ldots\}$$

and with instance the tree shown in Fig. 16.

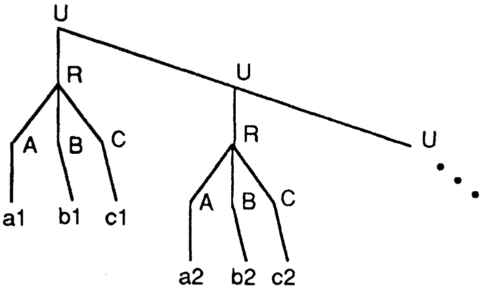


FIG. 16. *A representation of a flat relation instance in the grammatical model.*

We now consider three typical examples of unary relational algebra operators:

- The *renaming* of A to A' can be expressed as the child substitution $\Sigma\chi[R \rightarrow ABC, A, A']$.
- The *projection* of R onto AB can be expressed as the node deletion $\mathrm{N}\delta[C]$.
- The *selection* $A = C$ can be expressed by consecutively performing the following operations:
  1. the parent equality substitution $\Sigma\sigma[R \rightarrow ABC, A, C, R']$;
  2. the node deletion $\mathrm{N}\delta[R]$;
  3. the parent substitution $\Sigma\pi[R' \rightarrow ABC, R]$.

If desired, redundant U-nodes can be removed by repeatedly applying the node mergings $\mathrm{N}\mu[U \rightarrow U, U \rightarrow RU]$ and $\mathrm{N}\mu[U \rightarrow RU, U \rightarrow U]$.

Notice in the last example that the grammatical algebra does *not* have an iteration construct. The number of applications of the node mergings is therefore instance dependent (in this case, linear in the size of the instance).

4. A calculus for transforming information bases. Whereas in the previous section we defined a transformation language on information bases based on eight primitive operators which we claim to be sufficiently powerful to express a large class of queries, it is also possible to define a more declarative transformation language inspired by the relational calculus.

An expression in the grammatical calculus we propose consists of a set of conditions and a transformation clause, both built from variables. Informally, applying a calculus expression to an information base means performing the required transformations for each "occurrence" of the variables satisfying the set of conditions. Since the variables in a calculus expression represent so-called *rootless data trees*, we first explain this notion.

DEFINITION 4.1. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. A *rootless data tree* (R-tree) over $\mathcal{G}$ is a finite sequence of D-trees over $\mathcal{G}$. The set of all R-trees over $\mathcal{G}$ is denoted $\mathcal{R}(\mathcal{G})$.

We have to introduce the following notations concerning R-trees.

DEFINITION 4.2. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{R}$ be an R-tree over $\mathcal{G}$. Then

- $\mathbf{R}= (\mathbf{D}_1, \ldots, \mathbf{D}_n)$ denotes the sequence of D-trees of which $\mathbf{R}$ consists;
- $\mathbf{top}(\mathbf{R}) = (\mathbf{rt}(\mathbf{D}_1), \ldots, \mathbf{rt}(\mathbf{D}_n))$ denotes the sequence of the roots of the D-trees of which $\mathbf{R}$ consists;
- $\mathbf{F}$ denotes the empty R-tree.

Obviously, the isomorphism between D-trees defined in Definition 2.7 can be extended in a natural way to R-trees. The definition of R-trees also leads us to the following straightforward conclusions.

PROPOSITION 4.3. *Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme.*

- *Any interval*[2] *of an* R*-tree over $\mathcal{G}$ is also an* R*-tree over $\mathcal{G}$.*
- *Let* $\mathbf{D}$ *be a* D*-tree over $\mathcal{G}$. Then* $(\mathbf{D})$ *is an* R*-tree.*
- *Let* $\mathbf{D}$ *be a* D*-tree over $\mathcal{G}$. Then* $\mathbf{chltrs}(\mathbf{rt}(\mathbf{D}))$ *is an* R*-tree.*

It follows from Proposition 4.3 that R-trees can be contained in D-trees.

DEFINITION 4.4. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. Let $\mathbf{D}$ be a D-tree and $\mathbf{R}$ be an R-tree over $\mathcal{G}$. $\mathbf{R}$ is called a *rootless subtree* of $\mathbf{D}$ if $\mathbf{R}= (\mathbf{D}_1, \ldots, \mathbf{D}_n)$, $\mathbf{D}_1, \ldots, \mathbf{D}_n$ are subtrees of $\mathbf{D}$ and $\mathbf{top}(\mathbf{R})$ is a sequence of consecutive siblings of $\mathbf{D}$. The common parent of these siblings is denoted $\mathbf{par}(\mathbf{R})$. The set of all rootless subtrees of $\mathbf{D}$ is denoted $\mathbf{rst}(\mathbf{D})$. The R-tree $\mathbf{R}$ is called a *maximal rootless subtree* of $\mathbf{D}$ if, in addition, $\mathbf{top}(\mathbf{R})$ is a maximal sequence of consecutive siblings of $\mathbf{D}$, i.e., if $\mathbf{top}(\mathbf{R}) = \mathbf{chln}(\mathbf{par}(\mathbf{R}))$.

Finally, we need two simple operations on R-trees.

DEFINITION 4.5. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme.

- Let $\mathbf{R}_1 = (\mathbf{D}_1, \ldots, \mathbf{D}_m)$ and $\mathbf{R}_2 = (\mathbf{D}_{m+1}, \ldots, \mathbf{D}_n)$ be R-trees over $\mathcal{G}$. Then the *concatenation* $\mathbf{R}_1\mathbf{R}_2$ is the R-tree defined by $\mathbf{R}_1\mathbf{R}_2 = (\mathbf{D}_1, \ldots, \mathbf{D}_n)$.
- Let $\mathbf{R}$ be an R-tree over $\mathcal{G}$ and let $\mathbf{n}$ be an arbitrary node. If $\mathrm{lbl}(\mathbf{n})$ is an attribute or $\mathbf{R}$ is empty, then the *completion* $\mathbf{nR}$ is the D-tree defined by $\mathbf{rt}(\mathbf{nR}) = \mathbf{n}$ and $\mathbf{chltrs}(\mathbf{n}) = \mathbf{R}$.

As mentioned, variables in a calculus expression represent rootless subtrees of the information base instance under consideration. From these variables, terms are built using concatenation and completion. As a consequence of Proposition 4.3, these terms in turn represent R-trees. The set of conditions in a calculus expression consists of declarations of variables by terms and of equations between variables and terms; the substitution clause consists of a variable and the term by which that variable has to substituted. Of course, the variables in the substitution clause have to occur in the set of conditions. Before formalizing the syntax of a grammatical calculus expression, we clarify the concept by an example.

Example 4.6. Reconsider the information base of Example 2.6 and the query of Example 3.20, described in the grammatical algebra. This query can also be solved by the following

[2] By an interval of a sequence we mean a subsequence consisting of consecutive elements.

grammatical calculus expression:

$$\begin{aligned}[\rho_2 \leftarrow{}'' (\langle \mathit{Tit}\rangle \rho_4)''(\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Volume}\rangle \rho_9) \mid \\ \{\rho_1 &:= (\langle \mathit{Ref}\rangle \rho_2) \\ \rho_2 &:= (\langle \mathit{Auths}\rangle \rho_3)(\langle \mathit{Tit}\rangle \rho_4)(\langle \mathit{Source}\rangle \rho_5)(\langle \mathit{Year}\rangle \rho_6) \\ \rho_5 &:= (\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Issue}\rangle \rho_8) \\ \rho_8 &:= (\langle \mathit{Vol}\rangle \rho_9) : (\langle \mathit{Nr}\rangle \rho_{10})\}].\end{aligned}$$

Another similar grammatical calculus expression that has the same effect on information bases over the scheme of Example 2.6 is the following:

$$\begin{aligned}[\rho_2 \leftarrow{}'' (\langle \mathit{Tit}\rangle \rho_4)''(\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Volume}\rangle \rho_9) \mid \\ \{\rho_1 &:= (\langle \mathit{Ref}\rangle \rho_2) \\ \rho_2 &:= (\langle \mathit{Auths}\rangle \rho_3)(\langle \mathit{Tit}\rangle \rho_4)(\langle \mathit{Source}\rangle \rho_5)\rho_6 \\ \rho_5 &:= (\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Issue}\rangle \rho_8) \\ \rho_8 &:= (\langle \mathit{Vol}\rangle \rho_9) : (\langle \mathit{Nr}\rangle \rho_{10})\}].\end{aligned}$$

In both expressions, the four declarations in the right-hand side specify a "pattern" in the bibliographic information base; each time that pattern is found in the instance, it must be changed according to the substitution clause.

We now formally define the syntax of the grammatical calculus. Throughout this exposition, we assume that $\mathcal{V} = \{\rho_i \mid i \geq 1\}$ is an infinitely enumerable set of *variables*.

DEFINITION 4.7. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme.

- A *basic term* over $\mathcal{G}$ has one of the following three types:
  - –type 0: a $(a \in T)$;
  - –type 1: ρ_i $(\rho_i \in \mathcal{V})$;
  - –type 2: $(A\rho_i)$ $(A \in V, \rho_i \in \mathcal{V})$.
- A *term* over $\mathcal{G}$ is a finite sequence of basic terms over $\mathcal{G}$ that contains at most one basic term of type 1 and in which each variable and each attribute appears at most once. The empty term is denoted ε. The set of all variables occurring in a term t is denoted $\mathrm{var}(t)$.

DEFINITION 4.8. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme.

- A *declaration* over $\mathcal{G}$ has the form $\rho_i := t$ with $\rho_i \in \mathcal{V}$ and t a term over $\mathcal{G}$ in which ρ_i does not occur.
- An *equation* over $\mathcal{G}$ has the form $\rho_i = \rho_j$ with $\rho_i, \rho_j \in \mathcal{V}$.

DEFINITION 4.9. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. Let $\mathcal{D} = \{\rho_i := t_i \mid i \in I\}$, I a set of indices, be a finite set of declarations over $\mathcal{G}$ in which no variable appears in the left-hand side of more than one declaration and in the right-hand side of more than one declaration. Let $\mathrm{var}(\mathcal{D})$ denote the set of all variables occurring in $\mathcal{D}$. Consider the *associated directed graph* $\mathcal{G}(\mathcal{D})$ with set of nodes $\mathrm{var}(\mathcal{D})$ and set of edges $\{\rho_j \rightarrow \rho_k \mid j \in I \text{ and } \rho_k \in \mathrm{var}(t_j)\}$. $\mathcal{D}$ is called *hierarchical* if $\mathcal{G}(\mathcal{D})$ is a tree, and, furthermore, the root ρ_{root} has a declaration of the form $\rho_{\mathrm{root}} := (A\rho_l)$ for some $A \in V$ and $\rho_l \in \mathcal{V}$.

For a hierarchical set of declarations it makes sense to define the following.

DEFINITION 4.10. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathcal{D}$ be a hierarchical set of declarations over $\mathcal{G}$.

- Let $\rho_i \in \mathrm{var}(\mathcal{D})$ be a variable which is not the root of $\mathcal{G}(\mathcal{D})$. Then ρ_i is said to be *of type* 1 (*of type* 2) if the unique basic term containing ρ_i in the right-hand side of a declaration of $\mathcal{D}$ is of type 1 (of type 2).
- Let $\rho_i \in \mathrm{var}(\mathcal{D})$ be an arbitrary variable. The *depth* $\mathrm{d}(\rho_i)$ of ρ_i is recursively defined as follows, using the hierarchy in $\mathcal{G}(\mathcal{D})$:

1. If ρ_i is the root of $\mathcal{G}(\mathcal{D})$, then $\mathrm{d}(\rho_i) = 0$;
2. If ρ_i is of type 1 and ρ_j is the parent of ρ_i, then $\mathrm{d}(\rho_i) = \mathrm{d}(\rho_j)$;
3. If ρ_i is of type 2 and ρ_j is the parent of ρ_i, then $\mathrm{d}(\rho_i) = \mathrm{d}(\rho_j) + 1$.

Observe that, given a hierarchical set of declarations $\mathcal{D}$, the root of $\mathcal{G}(\mathcal{D})$ has no type.

Example 4.11. Consider the set of declarations in the right-hand side of the first expression in Example 4.6:

$$\begin{aligned}\{\rho_1 &:= (\langle \mathit{Ref}\rangle \rho_2)\\ \rho_2 &:= (\langle \mathit{Auths}\rangle \rho_3)(\langle \mathit{Tit}\rangle \rho_4)(\langle \mathit{Source}\rangle \rho_5)(\langle \mathit{Year}\rangle \rho_6)\\ \rho_5 &:= (\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Issue}\rangle \rho_8)\\ \rho_8 &:= (\langle \mathit{Vol}\rangle \rho_9) : (\langle \mathit{Nr}\rangle \rho_{10})\}.\end{aligned}$$

Obviously, this is a hierarchical set of declarations. Its associated tree is shown in Fig. 17.

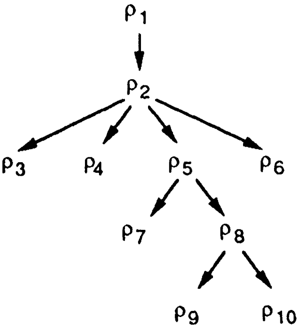


FIG. 17. *The tree associated with the hierarchical sets of declarations in Example* 4.11.

All variables (except of course for ρ_1) are of type 2. Furthermore, $\mathrm{d}(\rho_1) = 0$, $\mathrm{d}(\rho_2) = 1$, $\mathrm{d}(\rho_3) = \mathrm{d}(\rho_4) = \mathrm{d}(\rho_5) = \mathrm{d}(\rho_6) = 2$, $\mathrm{d}(\rho_7) = \mathrm{d}(\rho_8) = 3$, and $\mathrm{d}(\rho_9) = \mathrm{d}(\rho_{10}) = 4$. Now, consider the set of declarations in the right-hand side of the second expression in Example 4.6:

$$\begin{aligned}\{\rho_1 &:= (\langle \mathit{Ref}\rangle \rho_2)\\ \rho_2 &:= (\langle \mathit{Auths}\rangle \rho_3)(\langle \mathit{Tit}\rangle \rho_4)(\langle \mathit{Source}\rangle \rho_5)\rho_6\\ \rho_5 &:= (\langle \mathit{Journ}\rangle \rho_7)(\langle \mathit{Issue}\rangle \rho_8)\\ \rho_8 &:= (\langle \mathit{Vol}\rangle \rho_9) : (\langle \mathit{Nr}\rangle \rho_{10})\}.\end{aligned}$$

This is a hierarchical set of declarations with the same associated tree as the previous one. However, ρ_6 is now of type 1 with $\mathrm{d}(\rho_6) = 1$.

We now have all the ingredients to define an expression.

DEFINITION 4.12. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. An *expression* over $\mathcal{G}$ has the form $[\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ with $\rho_j \in \mathcal{V}$, u a term over $\mathcal{G}$ with $\rho_j \notin \mathrm{var}(u)$, $\mathcal{D}$ a hierarchical set of declarations over $\mathcal{G}$, and $\mathcal{E}$ a set of equations over $\mathcal{G}$, satisfying the following conditions:

1. All variables in the expression occur in $\mathcal{D}$;
2. No variable in u is an ancestor of ρ_j in $\mathcal{G}(\mathcal{D})$;
3. If, in addition, ρ_j is the root of $\mathcal{G}(\mathcal{D})$, then $u = (B\rho_k)$ for some $B \in V$ and $\rho_k \in \mathcal{V}$, or $u = \varepsilon$.

We invite the reader to check that the expression (without equations) in Example 4.6 satisfies Definition 4.12.

Before formally defining the semantics of a grammatical calculus expression, we show with examples how the grammatical algebra operators can be expressed in the calculus. As in the previous section, we are not concerned with the resulting information base schemes (about which we have not yet said anything with regard to the calculus).

Example 4.13. Reconsider Examples 3.4, 3.7, 3.9, and 3.15.

- The parent substitution $\Sigma\pi[A \to BA, E]$ can be expressed by

$$[\rho_1 \leftarrow (E\rho_2) \mid \{\rho_1 := (A\rho_2), \rho_2 := (B\rho_3)(A\rho_4)\}].$$

- The child substitution $\Sigma\epsilon[E \to BA, A, E]$ can be expressed by

$$[\rho_2 \leftarrow (B\rho_3)(E\rho_4) \mid \{\rho_1 := (E\rho_2), \rho_2 := (B\rho_3)(A\rho_4)\}].$$

- The child equality substitution $\Sigma\epsilon[E \to BE, E \to BE, B, D]$ can be expressed by

$$[\rho_2 \leftarrow (B\rho_3)(D\rho_4) \mid \{\rho_1 := (E\rho_2), \rho_2 := (B\rho_3)(E\rho_4), \rho_4 := (B\rho_5)(E\rho_6), \rho_3 = \rho_5\}].$$

- The node insertion $\text{N}\iota[B \to abEd, aFd]$ can be expressed by

$$[\rho_2 \leftarrow a(F\rho_3)d \mid \{\rho_1 := (B\rho_2), \rho_2 := a\rho_3 d, \rho_3 := b(E\rho_4)\}].$$

- The node deletion $\text{N}\delta[F]$ can be expressed by $[\rho_1 \leftarrow \varepsilon \mid \rho_1 := (F\rho_2)\}]$.
- The downward duplication $\Delta\delta[A \to BaC, B \to bd, C, D]$ can be expressed by

$$[\rho_3 \leftarrow bd(D\rho_4) \mid \{\rho_1 := (A\rho_2), \rho_2 := (B\rho_3)a(C\rho_4), \rho_3 := bd\}].$$

- The permutation $\Pi[B \to bBc, abB]$ can be expressed by

$$[\rho_2 \leftarrow ab(B\rho_3) \mid \{\rho_1 := (B\rho_2), \rho_2 := b(B\rho_3)c\}].$$

Describing the semantics of the grammatical calculus should consist of two parts: explaining what happens with schemes and explaining what happens with instances. Since, as observed earlier in this paper, it is unrealistic to compare information base operations at the scheme level with respect to expressiveness, we shall not elaborate on how calculus expressions work on information base schemes. The example below, however, should nevertheless convince the reader that calculus expressions can be applied to the schemes as well.

Example 4.14. Consider the calculus expression

$$[\rho_2 \leftarrow (D\rho_4) \mid \{\rho_1 := (A\rho_2), \rho_2 := (B\rho_3)(C\rho_4)\}]$$

and let $\mathcal{G} = (V, T, S, P)$ be an information base scheme to which this calculus expression is applied. Let $\mathcal{G}' = (V', T', S', P')$ denote the resulting scheme. Then $V' = V \cup \{D\}$ and $T' = T$ since there are no other attributes or constants in the substitution clause not occurring in one of the declarations. Furthermore, $S' = S$, since the attribute in the term defining the root variable is not altered. Finally, if $A \to BC \in P$, then $P' = P \cup \{A \to D\} \cup \{D \to s \mid C \to s \in P\}$, else $P' = P$.

Note that, for the sake of generality, the application of a calculus expression to an information base scheme can only result in *adding* new productions.

We now formally define the semantics of a grammatical calculus expression at the instance level. Although it is conceptually simple, as should be clear from the examples given thus far, the formalism itself is rather involved. This stems mainly from the fact that, when rearranging subtrees in applying a calculus expression, one must be able to describe how this rearrangement is "propagated" downward into these subtrees.

The evaluation of a calculus expression on a given information base instance can be described in two distinct stages.

1. First, the variables in an expression are "valuated" as rootless subtrees of the considered information base instance, satisfying the declarations and equations in that expression.

2. Then, the D-tree representing the information base instance is transformed according to these valuations and the transformation rule in the left-hand side of the expression.

The first stage is described in Definition 4.15, the second one in Definition 4.18.

DEFINITION 4.15. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $\mathcal{D}$ be a set of declarations and $\mathcal{E}$ a set of equations over $\mathcal{G}$ such that all variables in $\mathcal{E}$ occur in $\mathcal{D}$. Let $f\colon \text{var}(\mathcal{D}) \to \mathbf{rst}(\mathbf{D})$ be a total mapping from variables in $\mathcal{D}$ to rootless subtrees of $\mathbf{D}$. f is called a *valuation* of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$ if

1. for each declaration $\rho_i := \varepsilon$ in $\mathcal{D}$, $f(\rho_i) = \mathbf{F}$;

2. for each declaration $\rho_i := t$ in $\mathcal{D}$ with $t = t_1 \dots t_k$, $t_1, \dots, t_k$ being basic terms, $f(\rho_i) = \mathbf{R}_1 \dots \mathbf{R}_k$ with, for each $j = 1, \dots, k$,

    1. if $t_j = a$ for some $a \in T$, then $\mathbf{R}_j$ is a single-node rootless tree the node of which is labeled a,

    2. if $t_j = \rho_k$ for some $\rho_k \in \mathcal{V}$, then $\mathbf{R}_j = f(\rho_k)$,

    3. if $t_j = (A\rho_k)$ for some $A \in V$ and $\rho_k \in \mathcal{V}$, then there is a node $\mathbf{n}$ in $\mathbf{D}$ with $\text{lbl}(\mathbf{n}) = A$ such that $\mathbf{R}_j = (\mathbf{n} f(\rho_k))$,

3. for each equation $\rho_i = \rho_j$ in $\mathcal{E}$, $f(\rho_i) \cong f(\rho_j)$.

The set of all valuations of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$ is denoted $\mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$.

Example 4.16. Consider the information base instance $\mathbf{D}$ of Fig. 18 (over some appropriate scheme) and let E be the following calculus expression:

$$[\rho_5 \leftarrow (C\rho_7)\rho_8 \mid \{\rho_1 := (A\rho_2),\ \rho_2 := \rho_3 c(A\rho_4),\ \rho_3 := (B\rho_5)\rho_6,\ \rho_4 := \rho_7 c(A\rho_8),$$
$$\rho_5 := (C\rho_9)(D\rho_{10}),\ \rho_7 := (B\rho_{11})\rho_{12},\ \rho_{10} := \varepsilon,\ \rho_{11} := (C\rho_{13})\rho_{14},\ \rho_9 = \rho_{13}\}].$$

There are three valuations of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$. For each such valuation f, the node $\mathbf{n}$ satisfying $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$ has been marked by a square in Fig. 18. We leave it to the reader to check that these markings completely determine the corresponding valuations.

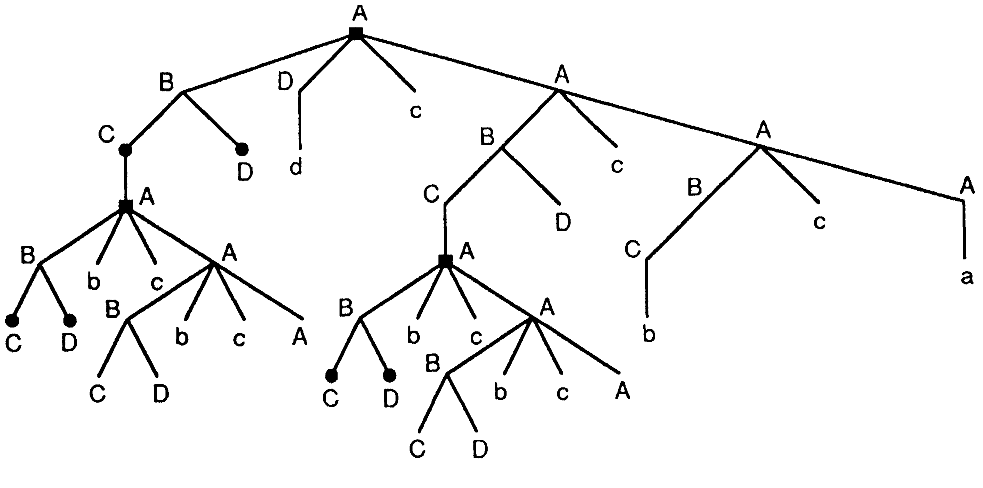


FIG. 18. *An information base instance* $\mathbf{D}$.

For later use, we used dots to mark all nodes in Fig. 18 that occur in $\mathbf{top}(f(\rho_5))$, for each valuation f of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$.

In Example 4.16, it turned out that the valuations under consideration can be completely characterized by indicating to which rootless subtree the root variable of the expression is mapped. Lemma 4.17 says even more: it suffices to specify a rootless subtree of which the valuation of one arbitrary variable is an interval. In particular, Lemma 4.17 ensures that the way in which $f(\rho_5)$ was indicated in Example 4.16 is unambiguous. More generally, it excludes that images of the same variables under different valuations "overlap." This will allow us to define the result of a calculus expression by performing a transformation on the information base instance under consideration for each valuation of its hierarchical set of declarations and set of equations.

LEMMA 4.17. *Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let $\mathcal{D}$ be a hierarchical set of declarations over $\mathcal{G}$, and let $\mathcal{E}$ be a set of equations over $\mathcal{G}$ such that all variables in $\mathcal{E}$ occur in $\mathcal{D}$. Let ρ_i be an arbitrary variable in $\mathcal{D}$, and let $\mathbf{R}$ be a rootless subtree of $\mathbf{D}$. There exists at most one valuation f of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$ such that $f(\rho_i)$ is an interval of $\mathbf{R}$.*

Proof. Suppose there exists a valuation f such that $f(\rho_i)$ is an interval of $\mathbf{R}$. We first show that for some variable $\rho_l \in \operatorname{var}(\mathcal{D})$, $f(\rho_l)$ is unambiguously determined by this condition. If ρ_i is the root of $\mathcal{G}(\mathcal{D})$, we know there exists a declaration in $\mathcal{D}$ of the form $\rho_i := (A\rho_j)$. Since $\mathbf{R}$ contains at most one subtree with a root labeled A, $f(\rho_i)$ is unambiguously determined. Now suppose ρ_i is *not* the root of $\mathcal{G}(\mathcal{D})$. Then there exists a sequence of variables in $\mathcal{D}$, say $\rho_{i_0}, \ldots, \rho_{i_k}$, $k \geq 1$, that satisfy the following conditions:

1. $\rho_{i_0} = \rho_i$;
2. For all $l = 1, \ldots, k-1$ there is a declaration in $\mathcal{D}$ of the form $\rho_{i_l} := \ldots \rho_{i_{l-1}} \ldots$;
3. There is a declaration in $\mathcal{D}$ of the form $\rho_{i_k} := \ldots (A\rho_{i_{k-1}}) \ldots$.

Note that the second condition is voidlessly satisfied if $k = 1$. The last condition can always be satisfied because a similar condition holds for the root of $\mathcal{G}(\mathcal{D})$. Now let $\mathbf{R}'$ be the unique maximal rootless subtree of $\mathbf{D}$ containing $\mathbf{R}$. Then, by condition 2 above, $f(\rho_i)$ being an interval of $\mathbf{R}$ implies that $f(\rho_{i_0}), \ldots, f(\rho_{i_{k-1}})$ are all intervals of $\mathbf{R}'$. Since a declaration of the form $\rho_{i_k} := (A\rho_{i_{k-1}})$ is in $\mathcal{D}$, it now follows that $f(\rho_{i_{k-1}})$ *equals* $\mathbf{R}'$.

Up to now, we have shown there exists some variable ρ_l in $\mathcal{D}$ for which $f(\rho_l)$ is a fully determined rootless subtree of $\mathbf{D}$. Since it is easily shown that whenever f is unambiguously determined for a certain variable it is also unambiguously determined for both the parent and all children of that variable, a straightforward induction shows that f is unambiguously determined for *all* variables in $\mathcal{D}$. □

We now define how an expression transforms D-trees. First, Definition 4.18 introduces so-called E-transformations. Theorem 4.19 then establishes the uniqueness of these E-transformations. Finally, Definition 4.20 points out how the unique E-transformation must be used to define the result of the calculus expression E.

We start with the notion of E-transformation. When rearranging rootless subtrees of a given D-tree in applying a calculus expression, we must be able to describe how the rearrangement is "propagated" downward into these subtrees. Intuitively, the resulting tree will therefore have to be constructed "bottom-up." Thus we cannot just define the effect of a calculus expression on D-trees alone; we need to define the effect on all rootless subtrees as well. The effect of a calculus expression on a rootless subtree is context-sensitive, however. Therefore, we introduce the notion of E-transformation, which defines the effect of a calculus expression E on an R-tree $\mathbf{R}$ in the context of a D-tree $\mathbf{D}$ of which $\mathbf{R}$ is a rootless subtree.

DEFINITION 4.18. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme and let $E \equiv [\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ be an expression over $\mathcal{G}$. A partial mapping $g\colon \mathcal{R}(\mathcal{G}) \times \mathcal{D}(\mathcal{G}) \rightarrow \mathcal{R}(\mathcal{G})/_{\cong}$ is an *E-transformation* of $\mathcal{G}$ if it satisfies the following conditions:[3]

[3] For reasons of convenience, the conditions are formulated as if $g(\mathbf{R}, \mathbf{D})$ were an arbitrary representation of the class under consideration.

1. $g(\mathbf{R}, \mathbf{D})$ is defined if and only if $\mathbf{R} \in \mathbf{rst}(\mathbf{D})$;
2. For all $\mathbf{D} \in \mathcal{D}(\mathcal{G})$, $g(\mathbf{F}, \mathbf{D}) \cong \mathbf{F}$;[4]
3. For some $\mathbf{D} \in \mathcal{D}(\mathcal{G})$, let $(\mathbf{nR}) \in \mathbf{rst}(\mathbf{D})$ be a rootless subtree such that for no valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$, $f(\rho_j) = (\mathbf{nR})$. Then $g((\mathbf{nR}), \mathbf{D}) \cong (\mathbf{n}g(\mathbf{R}, \mathbf{D}))$;
4. For some $\mathbf{D} \in \mathcal{D}(\mathcal{G})$, let $\mathbf{R} \in \mathbf{rst}(\mathbf{D})$ be a rootless subtree such that for no valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$, $f(\rho_j)$ is an interval of $\mathbf{R}$. Then if $\mathbf{R} = \mathbf{R}_1\mathbf{R}_2$, $g(\mathbf{R}, \mathbf{D}) \cong g(\mathbf{R}_1, \mathbf{D})g(\mathbf{R}_2, \mathbf{D})$;
5. For some $\mathbf{D} \in \mathcal{D}(\mathcal{G})$, let $\mathbf{R} \in \mathbf{rst}(\mathbf{D})$ be a rootless subtree such that for some valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$, $f(\rho_j) = \mathbf{R}$. Let $u = u_1 \ldots u_n$ with $u_1, \ldots, u_n$ basic terms. Then $g(\mathbf{R}, \mathbf{D}) \cong \mathbf{R}_1 \ldots \mathbf{R}_n$ with, for $k = 1, \ldots, n$,
    1. if $u_k = a$ for some $a \in T$, then $\mathbf{R}_k$ is a one-node rootless tree labeled a,
    2. if $u_k = \rho_l$ for some $\rho_l \in \mathcal{V}$, then we distinguish two cases:[5]
        1. if ρ_l is of type 1 in $\mathcal{D}$, and $f(\rho_l) = (\mathbf{D}_1, \ldots, \mathbf{D}_m)$, then
$$\mathbf{R}_k \cong g((\mathbf{D}_1), \mathbf{D}_1) \ldots g((\mathbf{D}_m), \mathbf{D}_m),$$
        2. if ρ_l is of type 2 in $\mathcal{D}$, and $\mathbf{n} = \mathbf{par}(f(\rho_l))$, then
$$\mathbf{R}_k \cong g(f(\rho_l), \mathbf{n}f(\rho_l));$$
    3. if $u_k = (A\rho_l)$ for some $A \in V$ and $\rho_l \in \mathcal{V}$, then we distinguish two cases:[6]
        1. if ρ_l is of type 1 in $\mathcal{D}$, and $f(\rho_l) = (\mathbf{D}_1, \ldots, \mathbf{D}_m)$, then $\mathbf{R}_k \cong (\mathbf{n}'\mathbf{R}')$ with $\mathrm{lbl}(\mathbf{n}') = A$ and $\mathbf{R}' \cong g((\mathbf{D}_1), \mathbf{D}_1) \ldots g((\mathbf{D}_m), \mathbf{D}_m)$,
        2. if ρ_l is of type 2 in $\mathcal{D}$, and $\mathbf{n} = \mathbf{par}(f(\rho_l))$, then $\mathbf{R}_k \cong (\mathbf{n}'\mathbf{R}')$ with $\mathrm{lbl}(\mathbf{n}') = A$ and $\mathbf{R}' \cong g(f(\rho_l), \mathbf{n}f(\rho_l))$;
6. For some $\mathbf{D} \in \mathcal{D}(\mathcal{G})$, let $\mathbf{R} = \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 \in \mathbf{rst}(\mathbf{D})$ be a rootless subtree such that for some valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$, $f(\rho_j) = \mathbf{R}_2$. Then $g(\mathbf{R}, \mathbf{D}) \cong g(\mathbf{R}_1, \mathbf{D})g(\mathbf{R}_2, \mathbf{D})g(\mathbf{R}_3, \mathbf{D})$.

We now establish the uniqueness of E-transformations.

THEOREM 4.19. *Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme and let $E \equiv [\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ be an expression over $\mathcal{G}$. Then there exists a unique E-transformation of $\mathcal{G}$.*

Proof. The proof goes by double induction. For the empty D-tree $\mathbf{E}$, we know that $g(\mathbf{F},\mathbf{E}) = \mathbf{F}$. We now assume that g is uniquely defined on all pairs $(\mathbf{R}, \mathbf{D})$ with $\mathbf{R}$ being a rootless subtree of $\mathbf{D}$ and the depth of $\mathbf{D}$ at most, say p (outer induction hypothesis). Now let $\mathbf{D}$ be a D-tree over $\mathcal{G}$ with depth $p + 1$. We know that $g(\mathbf{F}, \mathbf{D}) = \mathbf{F}$. We now also assume that g is uniquely defined on all pairs $(\mathbf{R}, \mathbf{D})$ with $\mathbf{R}$ being a rootless subtree of $\mathbf{D}$ and the depth[7] of $\mathbf{R}$ at most, say q (inner induction hypothesis). Now let $\mathbf{R}$ be a rootless subtree of $\mathbf{D}$ with depth $q + 1$. We distinguish two cases.

Case 1. *There is no valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$ for which $f(\rho_j)$ is an interval of* $\mathbf{R}$. Let $\mathbf{R} = (\mathbf{D}_1, \ldots, \mathbf{D}_n)$ with, for $i = 1, \ldots, n$, $\mathbf{D}_i = \mathbf{n}_i\mathbf{R}_i$ for some $\mathbf{R}_i \in \mathbf{rst}(\mathbf{D})$ with the depth of $\mathbf{R}_i$ at most q. Items 2, 3, and 4 of Definition 4.18 and the inner induction hypothesis guarantee that $g(\mathbf{R}, \mathbf{D})$ is uniquely defined.

Case 2. *For some valuation $f \in \mathcal{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{D})$, $f(\rho_j)$ is an interval of* $\mathbf{R}$. By Lemma 4.17, we know this f is unique. Let $\mathbf{R} = \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3$ with $\mathbf{R}_2 = f(\rho_j)$. By item 6 of Definition 4.18, $g(\mathbf{R}, \mathbf{D}) \cong g(\mathbf{R}_1, \mathbf{D})g(\mathbf{R}_2, \mathbf{D})g(\mathbf{R}_3, \mathbf{D})$. By the first case of this proof, $g(\mathbf{R}_1, \mathbf{D})$ and $g(\mathbf{R}_3, \mathbf{D})$ are uniquely determined. Let $u = u_1 \ldots u_n$ with $u_1, \ldots, u_n$ basic terms. By item 5 of

[4] Recall from Definition 4.2 that $\mathbf{F}$ is the empty R-tree.

[5] The reason for this distinction is the following. If a variable is part of a basic term of type 2, then we are interested in the entire tree this basic term represents, as opposed to merely the R-tree the variable represents.

[6] See footnote 5.

[7] The depth of $\mathbf{F}$ is 0; the depth of another R-tree is defined as the maximum of the depths of the D-trees of which it is composed.

Definition 4.18, we know that $g(\mathbf{R}_2, \mathbf{D}) \cong \mathbf{R}_{21} \dots \mathbf{R}_{2n}$ for some R-trees $\mathbf{R}_{21}, \dots, \mathbf{R}_{2n}$. It remains to show that, for $k = 1, \dots, n$, $\mathbf{R}_{2k}$ is uniquely determined. To do this we distinguish five subcases.

Subcase 1. *$u_k = a$ with $a \in T$.* By item 5.1 of Definition 4.18, $\mathbf{R}_{2k}$ is unambiguously determined.

Subcase 2. *$u_k = \rho_l$ with ρ_l a type* 1 *variable in $\mathcal{D}$.* Let $f(\rho_l) = (\mathbf{D}_1, \dots, \mathbf{D}_m)$. By item 5.2.1 of Definition 4.18, we know that $\mathbf{R}_{2k} \cong g((\mathbf{D}_1), \mathbf{D}_1) \dots g((\mathbf{D}_m), \mathbf{D}_m)$. Since, by Definition 4.12, the depth of ρ_l in $\mathcal{G}(\mathcal{D})$ is at least 1, it follows that, for all $j = 1, \dots, m$, the depth of $\mathbf{D}_j$ is at most p. Hence the desired conclusion follows from the outer induction hypothesis.

Subcase 3. *$u_k = \rho_l$ with ρ_l a type* 2 *variable in $\mathcal{D}$.* Let $\mathbf{n} = \mathbf{par}(f(\rho_l))$. By item 5.2.2 of Definition 4.18, we know that $\mathbf{R}_{2k} \cong g(f(\rho_l), \mathbf{n}f(\rho_l))$. If $\mathbf{n}$ is *not* the root of $\mathbf{D}$, then the depth of $\mathbf{n}f(\rho_l)$ is at most p. Hence the desired conclusion follows from the outer induction hypothesis. However, if $\mathbf{n} = \mathbf{rt}(\mathbf{D})$, then $\mathbf{D} = \mathbf{n}f(\rho_l)$, whence the depth of $f(\rho_l)$ is at most p. Moreover, by Definition 4.12, it follows that $f(\rho_j) = (\mathbf{D}) = \mathbf{R}_2 = \mathbf{R}$, whence $n = k = 1$ and $p = q$. So, $f(\rho_l)$ has also depth at most q. The uniqueness of $\mathbf{R}_{2k} = \mathbf{R}_{21}$ now follows from the inner induction hypothesis.

Subcase 4. *$u_k = (A\rho_l)$ with $A \in V$ and ρ_l a type* 1 *variable in $\mathcal{D}$.* Let $f(\rho_l) = (\mathbf{D}_1, \dots, \mathbf{D}_m)$. By item 5.3.1 of Definition 4.18, we know that $\mathbf{R}_{2k} \cong (\mathbf{n}'\mathbf{R}')$ with $\mathrm{lbl}(\mathbf{n}') = A$ and $\mathbf{R}' \cong g((\mathbf{D}_1), \mathbf{D}_1) \dots g((\mathbf{D}_m), \mathbf{D}_m)$. The remainder of this case is now analogous to subcase 2.

Subcase 5. *$u_k = (A\rho_l)$ with $A \in V$ and ρ_l a type* 2 *variable in $\mathcal{D}$.* Let $\mathbf{n} = \mathbf{par}(f(\rho_l))$. By item 5.3.2 of Definition 4.18, we know that $\mathbf{R}_{2k} \cong (\mathbf{n}'\mathbf{R}')$ with $\mathrm{lbl}(\mathbf{n}') = A$ and $\mathbf{R}' \cong g(f(\rho_l), \mathbf{n}f(\rho_l))$. The remainder of this case is now analogous to subcase 3. □

Using the unique transformation defined above, we finally define the result of a calculus expression.

DEFINITION 4.20. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme and let $\mathbf{D}$ be an information base instance over $\mathcal{G}$. Let E be an expression over $\mathcal{G}$. Let g be the unique E-transformation of $\mathcal{G}$. Then $E(\mathbf{D})$ is (the class of) the D-tree $\mathbf{D}'$ for which $g((\mathbf{D}), \mathbf{D}) \cong (\mathbf{D}')$, if this D-tree is in turn an information base instance, and undefined otherwise.

We conclude this section with a final example.

Example 4.21. Reconsider the information base instance $\mathbf{D}$ and the calculus expression E of Example 4.16. Recall that in Fig. 18, for each valuation f, the (unique) node in $\mathbf{top}(f(\rho_1))$ is marked by a square and all nodes in $\mathbf{top}(f(\rho_5))$ are marked by dots (ρ_1 is the root of the expression and ρ_5 is the left-hand side of the substitution clause). The reader is invited to check that the result $E(\mathbf{D})$ of applying the calculus expression E to the information base instance $\mathbf{D}$ indeed equals the instance of Fig. 19.

5. Properties of the grammatical calculus. The definition of the grammatical calculus given in the previous section raises several decidability issues. Since a full treatment of these decidability issues would go beyond the scope of the present article, we shall deal with these here only briefly, in a fairly informal manner.

The key construct of this section is that of a *condition tree* of a calculus expression, or more precisely, of the set of hierarchical declarations and equations of a calculus expression. We first define this notion for the case in which no equations are present.

DEFINITION 5.1. Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $\mathcal{D}$ be a hierarchical set of declarations.

- The condition tree $\mathbf{C}(\mathcal{D})$ of $\mathcal{D}$ is constructed as follows. First, initialize $\mathbf{C}(\mathcal{D})$ to a tree consisting of one node labeled with the root of $\mathcal{G}(\mathcal{D})$. Then, as long as there is a node $\mathbf{n}$ in $\mathbf{C}(\mathcal{D})$ labeled with a variable ρ_i that is not a leaf in $\mathcal{G}(\mathcal{D})$, i.e., for which there exists some

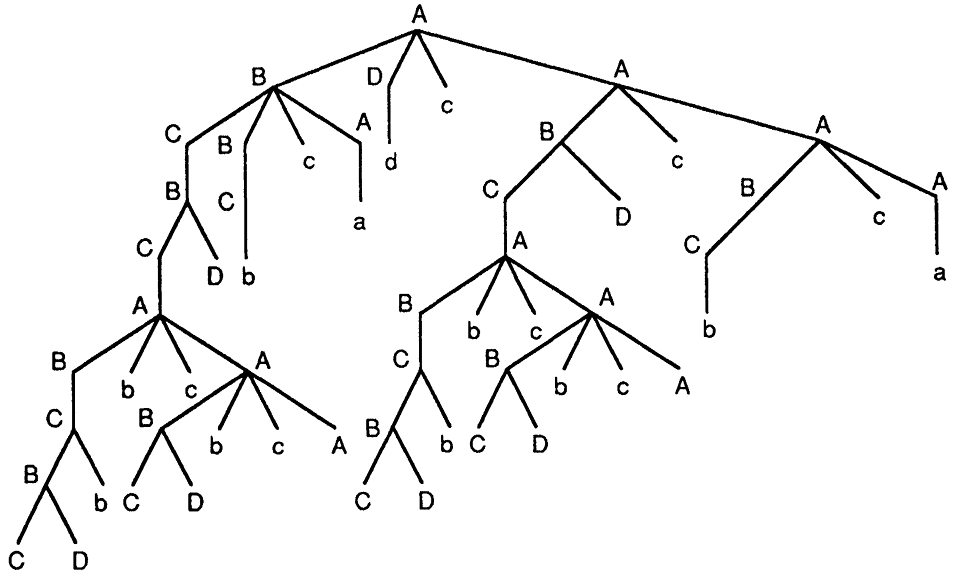


FIG. 19. *The information base instance $E(\mathbf{D})$.*

hierarchical declaration $\rho_i := t_1 \ldots t_m$ in $\mathcal{D}$, substitute $\mathbf{n}$ by the rootless subtree $(\mathbf{C}_1, \ldots, \mathbf{C}_m)$ where, for $k = 1, \ldots, m$,

1. if $t_k = a$ with $a \in T$, $\mathbf{C}_k$ consists of one node which is labeled a;

2. if $t_k = \rho_l$ with ρ_l a type 1 variable in $\mathcal{D}$, $\mathbf{C}_k$ consists of one node which is labeled ρ_l;

3. if $t_k = (A\rho_l)$ with $A \in V$ and ρ_l a type 2 variable in $\mathcal{D}$, $\mathbf{C}_k$ is a two-node tree the root of which is labeled A and the leaf of which is labeled ρ_l.

- Let $E \equiv [\rho_j \leftarrow u \mid \mathcal{D}]$ be an expression over $\mathcal{G}$ without equations. Then $\mathbf{C}(E) = \mathbf{C}(\mathcal{D})$.

Notice that $\mathbf{C}(\mathcal{D})$ in Definition 5.1 above is indeed a tree since the right-hand side of the declaration for the root in $\mathcal{D}$ consists of only one basic term, which is of type 2.

Example 5.2. Let $\mathcal{D}$ be the set of hierarchical declarations in Example 4.16. The condition tree $\mathbf{C}(\mathcal{D})$ of $\mathcal{D}$ is shown in Fig. 20.

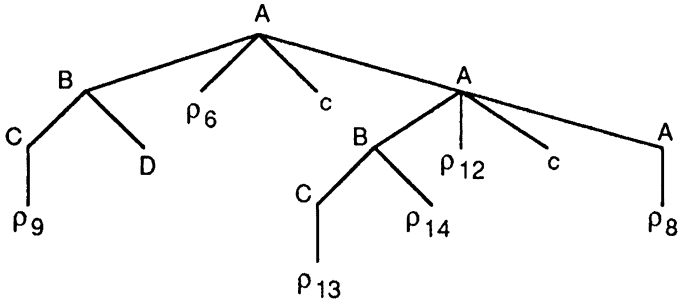


FIG. 20. *The condition tree $\mathbf{C}(\mathcal{D})$.*

Intuitively, the condition tree of an expression shows the pattern that must be present in an information base instance for the expression to have an action on that instance.

The notion of condition tree can be extended to the case in which both hierarchical declarations and equations are present. We shall informally explain how. Thereto, suppose that in addition to the assumptions of Definition 5.1 we have a set of equations $\mathcal{E}$ over $\mathcal{G}$.

If the equations only involve variables that are leaves in $\mathbf{C}(\mathcal{D})$ and hence also in $\mathcal{G}(\mathcal{D})$, then the condition tree $\mathbf{C}(\mathcal{D} \cup \mathcal{E})$ of $\mathcal{D}$ and $\mathcal{E}$ is straightforwardly constructed from $\mathbf{C}(\mathcal{D})$ by equating labels according to the equations in $\mathcal{E}$. For instance, if E is the expression in Example 4.16, then $\mathbf{C}(E)$[8] is obtained from the condition tree in Example 5.2, Fig. 20, by equating ρ_9 and ρ_{13}.

In the perhaps more pathological case in which variables are equated that are not necessarily leaves of $\mathbf{C}(\mathcal{D})$, the construction is somewhat more involved. To illustrate this, let $\mathcal{D}$ be the hierarchical set of declarations in the expression of Example 4.16 and consider the equation $\rho_5 = \rho_{11}$. The rootless subtrees corresponding to ρ_5 and ρ_{11} in $\mathbf{C}(\mathcal{D})$ in Example 4.16 are shown in Fig. 21 left and right, respectively.

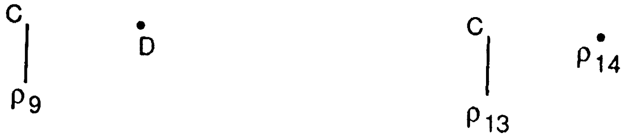


FIG. 21. *The rootless subtrees corresponding to ρ_5 and ρ_{11}.*

We can now try to expand in both rootless subtrees the nodes corresponding to variables in a minimal way such that the R-trees become isomorphic. In our example, this can be achieved by substituting ρ_{14} by D and equating ρ_9 and ρ_{13}. The R-tree thus obtained is actually the most general unifier of ρ_5 and ρ_{11}. Finally, the condition tree $\mathbf{C}(\mathcal{D} \cup \{\rho_5 = \rho_{11}\})$ of $\mathcal{D}$ and $\{\rho_5 = \rho_{11}\}$ is obtained by substituting ρ_5 and ρ_{11} by their most general unifier.

Of course this most general unifier need not exist, e.g., because the two rootless subtrees involved are incompatible. Also, the unification process might result in an infinite tree. The latter case would occur if we tried to compute $\mathbf{C}(\mathcal{D} \cup \{\rho_2 = \rho_4\})$. During the unification process we would find that ρ_8 must be equated to an R-tree strictly containing ρ_8 as a rootless subtree, whence the resulting tree would be infinite. Finally, it is possible that the resulting tree is not a *legal* D-*tree* in the sense that it contains sibling nodes labeled by the same attribute, whence the tree cannot be considered as an information base instance over some scheme.

Each time the construction of a condition tree requires an impossible unification process or a unification process resulting in an infinite tree, or does not result in a legal D-tree, we say that that the condition tree is *undefined.* Obviously, this property is decidable. The undefinedness of a condition tree corresponds to the fact that any associated expression is not applicable to any information base instance.

Several decidability results regarding the grammatical calculus can be proved by using condition trees. The techniques employed in these proofs in essence come down to applying expressions to their own condition tree and are therefore reminiscent of similar techniques used in the relational model for conjunctive queries [6], [21].

The first decidability result is concerned with checking whether or not a calculus expression represents the identity.

LEMMA 5.3. *Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, and let $E \equiv [\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ be an expression over $\mathcal{G}$. Then E represents the identity if and only if either the condition tree of E is undefined or $E(\mathbf{C}(E)) \cong \mathbf{C}(E)$.*

Proof. Obviously, if E represents the identity and the condition tree of E is defined, then $E(\mathbf{C}(E)) \cong \mathbf{C}(E)$, whence the "only if." To see the "if," we need to distinguish two cases. If

[8] If $E \equiv [\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ is an arbitrary expression over $\mathcal{G}$, then $\mathbf{C}(E) = \mathbf{C}(\mathcal{D} \cup \mathcal{E})$.

the condition tree of E is undefined, then E obviously represents the identity, since it is not applicable to any information base instance. Thus suppose the condition tree of E does exist and satisfies $E(\mathbf{C}(E)) \cong \mathbf{C}(E)$. Let f be the valuation[9] of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{C}(E)$. If f is extended to terms in the natural way, then $E(\mathbf{C}(E)) \cong \mathbf{C}(E)$ is equivalent to $f(\rho_j) \cong f(u)$. Using this latter condition, the proof can be completed by a straightforward double induction, as in the proof of Theorem 4.19. □

Using Lemma 5.3, we can prove the following necessary condition for two expressions to be equivalent.

THEOREM 5.4. *Let $\mathcal{G} = (V, T, S, P)$ be an information base scheme. Two expressions over $\mathcal{G}$ are equivalent only if either they both represent the identity or their condition trees are isomorphic upon renaming of variables.*

Proof. Let $E_1 \equiv [\rho_j^1 \leftarrow u_1 \mid \mathcal{D}_1 \cup \mathcal{E}_1]$ and $E_2 \equiv [\rho_j^2 \leftarrow u_2 \mid \mathcal{D}_2 \cup \mathcal{E}_2]$ be two equivalent expressions that do not represent the identity. By Lemma 5.3, the condition trees of E_1 and E_2 exist and satisfy $E_1(\mathbf{C}(E_2)) \not\cong \mathbf{C}(E_1)$ and $E_2(\mathbf{C}(E_2)) \not\cong \mathbf{C}(E_2)$. Hence, by the equivalence of E_1 and E_2, $E_2(\mathbf{C}(E_1)) \not\cong \mathbf{C}(E_1)$ and $E_1(\mathbf{C}(E_2)) \not\cong \mathbf{C}(E_2)$. In particular, this implies there must exist a valuation of $\mathcal{C}_1$ and $\mathcal{D}_1$ in $\mathbf{C}(E_2)$ as well as a valuation of $\mathcal{C}_2$ and $\mathcal{D}_2$ in $\mathbf{C}(E_1)$. Using this fact, the theorem is now easily shown. □

Given Theorem 5.4, it is now tempting to conjecture that two grammatical calculus expressions that do not represent the identity are equivalent if and only if they yield the same result when applied to their common condition tree. Unfortunately, this condition is not sufficient because of the special way in which type 2 variables in a substitution term are handled in the calculus. Example 5.5 gives a counterexample.

Example 5.5. Let $\mathcal{D} \cup \mathcal{E}$ be the following set of hierarchical declarations and equations (over some appropriate scheme):

$$\{\rho_1 := (A\rho_2),\ \rho_2 := (B\rho_3)(A\rho_4)(C\rho_5),\ \rho_3 := (A\rho_6),\ \rho_6 := \rho_7 b,\ \rho_4 = \rho_7\}.$$

The condition tree of $\mathcal{D}$ and $\mathcal{E}$ is shown in Fig. 22.

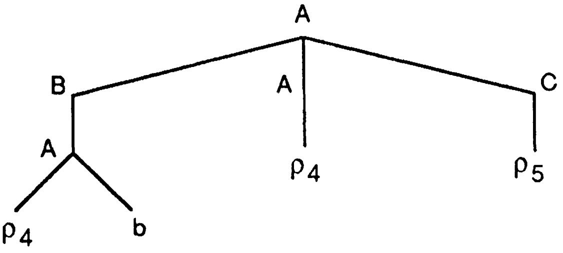


FIG. 22. *The condition tree* $\mathbf{C}(\mathcal{D} \cup \mathcal{E})$.

Now consider the expressions

$$E_1 \equiv [\rho_5 \leftarrow \rho_4 \mid \mathcal{D} \cup \mathcal{E}];$$
$$E_2 \equiv [\rho_5 \leftarrow \rho_7 \mid \mathcal{D} \cup \mathcal{E}].$$

Notice that ρ_4 is of type 2 while ρ_7 is of type 1.

Obviously, $E_1(\mathbf{C}(\mathcal{D}\cup\mathcal{E})) \cong E_2(\mathbf{C}(\mathcal{D}\cup\mathcal{E}))$, because these resulting trees are both obtained by substituting ρ_5 by ρ_4 in the tree of Fig. 22. Nevertheless, E_1 and E_2 are *not* equivalent. To see this, we replace the nodes in Fig. 22 that are labeled ρ_4 with the rootless subtree obtained

[9] By the definition of condition tree, there is a unique valuation $f \in \mathrm{F}(\mathcal{D} \cup \mathcal{E}, \mathbf{C}(E))$ determined by $f(\rho_{\mathrm{root}}) = (\mathbf{C}(E))$, ρ_{root} being the root of $\mathrm{G}(\mathcal{D})$.

by chopping off the root from a copy of the condition tree. The result of this modification is shown in Figure 23.

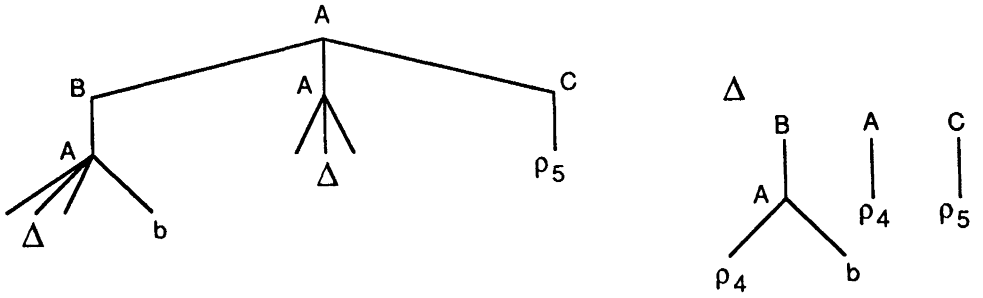


FIG. 23. *The modified condition tree.*

The purpose of this modification was substituting the tree rooted in the parent of ρ_4 by its most general unifier with a copy of the entire condition tree.

Applying expression E_2 to the tree in Fig. 23 results in the straightforward substitution of ρ_5 by the rootless subtree identified by Δ, as shown in Fig. 24 bottom.

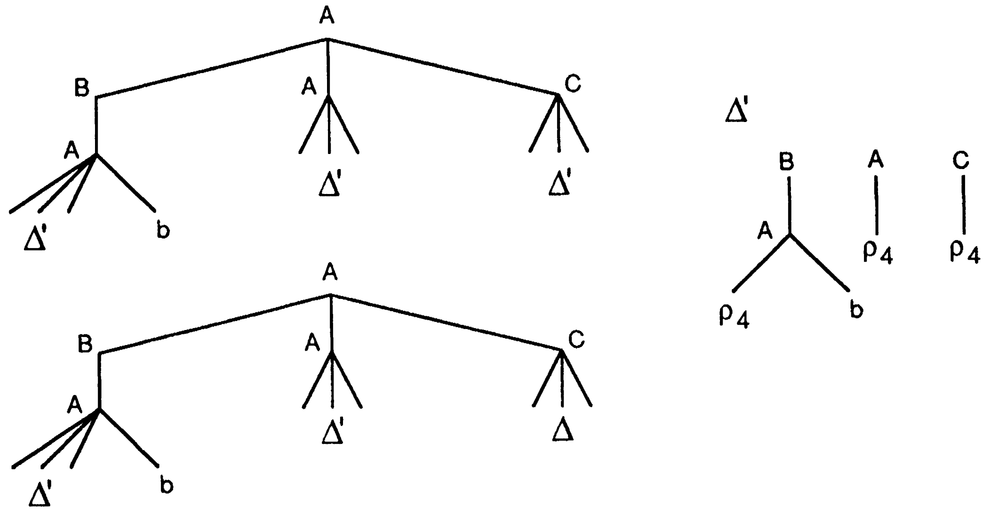


FIG. 24. *The results of applying* E_1 (top), *respectively,* E_2 (bottom), *to the tree of Fig.* 23.

In contrast, the application of E_1 results in the substitution of ρ_5 by Δ' where Δ' is obtained from Δ by applying E_1 to the tree rooted in the parent of ρ_4. This asymmetry is due to the fact that ρ_4 is of type 2, while ρ_7 is of type 1. The result of applying E_1 to the tree in Fig. 23 is shown in Fig. 24 top.

In general, given an expression and its condition tree, one has two choices for each type 2 variable in the substitution tree: one can either leave the corresponding node or rootless subtree in the condition tree unchanged, or one can transform it in the sense of Example 5.5. This procedure leads to a number of trees that is potentially exponential in the number of type 2 variables in the substitution term.[10] To facilitate our further discussion, we shall

[10] In most cases, however, the number of trees obtained will be significantly smaller because the transformation described in Example 5.5 requires a unification whose result may well be undefined.

call the set of legal D-trees thus obtained the *set of representative instances* of the given expression.

Intuitively, the set of representative instances of a calculus expression is constructed in such a way that for every possible valuation of the expression's set of declarations and equations in a concrete information base instance, there is a representative instance whose transformation by the expression "models" the way in which the information base instance is transformed locally.

Therefore, we conjecture that equivalence of nonidentity expressions can be decided by considering all trees in their sets of representative instances and verifying whether both expressions yield the same results for all those trees.

Finally, the set of representative instances can also be used to decide whether or not the result of a calculus expression is always defined, independent of the information base instance to which the expression is applied. Now, the result of a calculus expression applied to a concrete information base instance can only be undefined if the resulting tree is no longer an instance, i.e., if this tree contains sibling nodes labeled with the same attribute in V. By what has been said above, it suffices to apply the calculus expression to all representative instances to verify whether or not undefinedness can occur. Thereto, one has to check whether or not

1. one of the resulting trees contains sibling nodes labeled by the same attribute;
2. in one of the resulting trees it is possible to substitute a variable by a sibling attribute.

The latter case occurs precisely when a variable in a resulting instance has a sibling attribute that is not a sibling to that variable in the original representative instance. Hence we have the following theorem.

THEOREM 5.6. *It is decidable whether or not the result of a calculus expression over a given information base scheme is always defined.*

6. The equivalence between algebra and calculus. In §3, we presented the grammatical algebra as a query language for transforming information bases. In §§4 and 5, we introduced and discussed grammatical calculus expressions. We can now consider the grammatical calculus as the language consisting of all finite sequences of calculus expressions. Note that, in contrast to the relational calculus, we cannot hope such a sequence will always be equivalent to a single expression, since in general there is no way to combine the various condition trees of the expressions in the sequence into one single condition tree that could be used to describe the net effect of the transformation. Since each grammatical algebra operation can be expressed by a single calculus expression, the grammatical calculus will nevertheless allow a more succinct representation of queries than the algebra.

In this section, we compare the expressive power of the grammatical algebra and calculus. Inspired by the classical result in the relational model, we were able to prove their equivalence. In view of the technical complexity of this proof, we use Example 4.16 as running example throughout the proof in order to improve its readability.

THEOREM 6.1. *The grammatical algebra and grammatical calculus are equivalent with regard to expressive power.*

Proof. In Example 4.13, applications of all algebra operators (except for upward duplication, which is analogous to downward duplication) are expressed in the calculus. It is straightforward to generalize the techniques used in these examples. Hence the algebra can be simulated in the calculus. The more involved part of the proof consists of showing that a calculus expression can be simulated in the algebra. In order to show this, we shall simulate in the algebra the various steps needed to evaluate a calculus expression of which, without loss of generality, we assume it does not represent the identity. (This assumption is needed to guarantee, by Lemma 5.3, the existence of the expression's condition tree, which in turn is needed to validate some of the constructions made below.)

Therefore, let $\mathcal{G} = (V, T, S, P)$ be an information base scheme, let $\mathbf{D}$ be an instance over $\mathcal{G}$, and let $[\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$ be a calculus expression with $\mathrm{var}(\mathcal{D}) = \{\rho_1, \ldots, \rho_n\}$ and $\mathcal{D} = \{\rho_i := t_i \mid i \in I\}, I \subseteq \{1, \ldots, n\}$. Without loss of generality, we assume that ρ_k being an ancestor of ρ_l in $\mathcal{G}(\mathcal{D})$ implies $k < l$. (Observe that Example 4.16 satisfies this requirement.) We further assume that ρ_1 is the root of $\mathcal{G}(\mathcal{D})$ and that the unique declaration for ρ_1 in $\mathcal{D}$ has the form $\rho_1 = (A\rho_2)$.

We also number the equations starting from $n+1$: $\mathcal{E} = \{e_{n+1}, \ldots, e_{n+l}\}$. Since, obviously, a nontrivial equation involving ρ_1 can never be satisfied, we may assume, again without loss of generality, that ρ_1 is not contained in an equation of $\mathcal{E}$.

Let J be an arbitrary set of nonnegative integers. For each $B \in V$, we assume that B^J denotes an attribute; similarly, for each $a \in T$, we assume that a^J denotes a constant. We also assume that N_i, $i = 1, 2, 3, \ldots$, are attributes not in V. Finally, we also assume that N_i^J denotes an attribute. Informally speaking, the superscripts of the labels will be used to remember which variables can be valuated into which rootless subtrees. The N_i^J are auxiliary attributes which will be used for copying information in $\mathbf{D}$ from one place to another in the tree.

The proof is basically a construction that consists of the following steps:

Step 1. *Initialization.* We index all node labels in $\mathbf{D}$ with the empty set. This is done by using parent substitution (for the attribute nodes) and permutation (for the constant nodes).

Step 2. *Determination of all valuations of $\mathcal{D}$ in* $\mathbf{D}$. We shall relabel by $A^{\{1\}}$ all nodes $\mathbf{n}$ for which there exists a valuation[11] f of $\mathcal{D}$ in $\mathbf{D}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$. (Remember that A is the attribute in the declaration for the root ρ_1 of $\mathcal{G}(\mathcal{D})$.) Therefore, we do the following steps:

Substep 1. *Transforming $\mathcal{D}$.* From $\mathcal{D}$ we construct $\mathcal{D}'$ as follows. $\mathcal{D}'$ contains $\rho_1 := (A\rho_2)$ as well as one declaration for each type 2 variable. The right-hand side of this declaration contains only type 2 variables and type 1 variables that are leaves in $\mathcal{G}(\mathcal{D})$. These right-hand sides are obtained from the original right-hand sides in $\mathcal{D}$ by subsequent substitutions. For example, if $\mathcal{D}$ is the set of declarations in Example 4.16, then

$$\mathcal{D}' = \{\rho_1 := (A\rho_2),\ \rho_2 := (B\rho_5)\rho_6 c(A\rho_4),\ \rho_4 := (B\rho_{11})\rho_{12} c(A\rho_8),$$
$$\rho_5 := (C\rho_9)(D\rho_{10}),\ \rho_{10} := \varepsilon,\ \rho_{11} := (C\rho_{13})\rho_{14}\}.$$

Note that $\mathcal{D}'$ actually describes the structure of the condition tree $\mathbf{C}(\mathcal{D})$ (see Definition 5.1).[12] Obviously, the restriction to $\mathrm{var}(\mathcal{D}')$ of a valuation of $\mathcal{D}$ in $\mathbf{D}$ is a valuation of $\mathcal{D}'$ in $\mathbf{D}$; conversely, each valuation of $\mathcal{D}'$ in $\mathbf{D}$ can be extended to a valuation of $\mathcal{D}$ in $\mathbf{D}$.

Substep 2. *Indicating all rootless subtrees in* $\mathbf{D}$ *to which type* 2 *leaf nodes in $\mathcal{G}(\mathcal{D})$ can be mapped.* We shall indicate these rootless subtrees by adding to the superscripts of the labels of their parent nodes the indices of the corresponding variables. Thereto, we perform, in any order, the following operation for each type 2 leaf node ρ_i in $\mathcal{G}(\mathcal{D}')$, until no further action is possible. If there is no declaration $\rho_i := \varepsilon$ in $\mathcal{D}'$, we do $\Sigma\pi[B^J \to s, B^{J\cup\{i\}}]$ for each $B^J \to s$ with $i \notin J$ and either $B^J \to s$ a production in the current scheme or B^J an attribute in the current scheme and $s = \varepsilon$; if $\rho_i := \varepsilon$ is in $\mathcal{D}'$, we only do $\Sigma\pi[B^J \to \varepsilon, B^{J\cup\{i\}}]$.

In our example, ρ_{10} is the only type 2 leaf node in $\mathcal{G}(\mathcal{D})$ that has a declaration with an empty right-hand side. Consequently, an index 10 must be added to the D-labels of leaf nodes in the current instance: there are six such nodes. The other type 2 leaf nodes of $\mathcal{G}(\mathcal{D})$ are ρ_8, ρ_9 and ρ_{13}. Consequently, an index 8 must be added to all A-labels and indices 9 and 13 to all C-labels of nodes in the current instance. The result of these operations is shown in Fig. 25.

[11] For the time being, we ignore the equations in $\mathcal{E}$.

[12] Since by assumption the condition tree $\mathbf{C}(\mathcal{D})$ exists, it follows that the substitutions performed cannot yield illegal terms, i.e., terms containing two basic terms of type 2 with the same attribute.

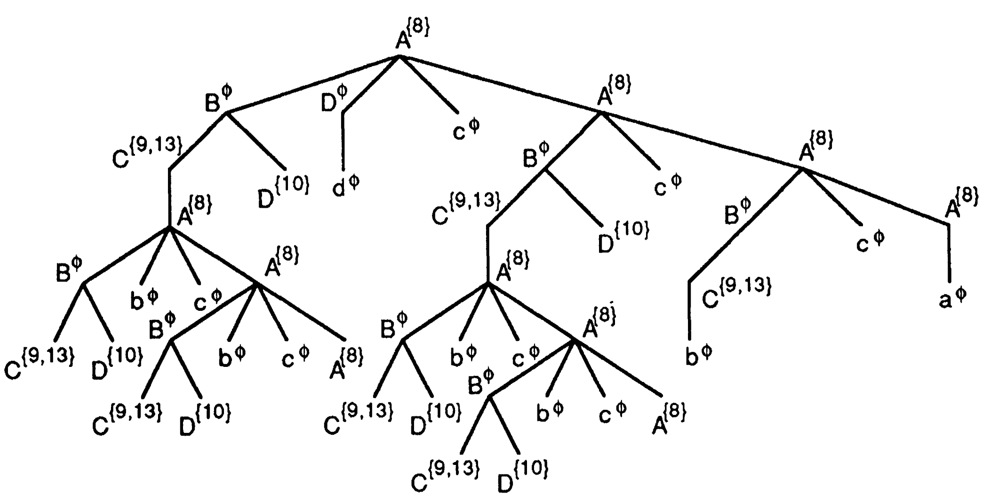


FIG. 25. *The result of marking all rootless subtrees in* **D** *to which type 2 leaf nodes in* $\mathcal{G}(\mathcal{D})$ *can be mapped.*

Substep 3. *Iteratively building up all valuations of* $\mathcal{D}$ *in* **D**. Let $\rho_{i_1}, \ldots, \rho_{i_m}$ be (in ascending order) all type 2 variables in $\mathcal{D}'$. Let, for $1 \le p \le m$, $\mathcal{D}'_p$ be the set of all declarations of $\mathcal{D}'$ involving only variables whose index is at least i_p. We will relabel the nodes in the current instance in such a way that, for each $p = 1, \ldots, m$,

the superscript of the label of a node **n** *contains the index* i_p *if and only if there exists a valuation* f *of* $\mathcal{D}'_p$ *in* **D** *with* $\mathbf{par}(f(\rho_{i_p})) = \mathbf{n}$ (condition (p)).

Note that, by the construction in the previous step, the current instance already satisfies all conditions (p) for which ρ_{i_p} is a leaf node in $\mathcal{G}(\mathcal{D}')$. By a downward iterative procedure, we now enforce the conditions (p) for which ρ_{i_p} is *not* a leaf node in $\mathcal{G}(\mathcal{D}')$. Thereto, we perform the following operation for those $p := m$ down to 1 for which ρ_{i_p} is an internal node of $\mathcal{G}(\mathcal{D}')$. Let $\rho_{i_p} := t_1 \ldots t_l$ be the declaration for ρ_{i_p} in $\mathcal{D}'$ with the t_q, $q = 1, \ldots, l$, basic terms. Let $\rho_k := \ldots (B\rho_{i_p}) \ldots$ be the declaration in $\mathcal{D}'$ containing ρ_{i_p} in its right-hand side. Then, in any order and until no further action is possible, we do $\Sigma\pi[B^J \to s_1 \ldots s_l,\ B^{J \cup \{i_p\}}]$ for each production $B^J \to s_1 \ldots s_l$ in the current scheme in which $i_p \notin J$ and, for $q = 1, \ldots, l$, the s_q have the following form:

$$s_q = \begin{cases} a^{\emptyset} & \text{if } t_q = a,\, a \in T; \\ C^K & \text{if } t_q = (C\rho_r),\, C \in V,\, \rho_r \in \operatorname{var}(\mathcal{D}'), \text{ and } r \in K \end{cases}$$

(s_q is arbitrary if t_q is of type 1).

The current instance for our example is now as shown in Fig. 26.

Observe that, by necessity, $i_1 = 2$. Hence the superscript of the label of a node **n** contains the index 2 if and only if there exists a valuation f of $\mathcal{D}' - \{\rho_1 \leftarrow (A\rho_2)\}$ in **D** with $\mathbf{par}(f(\rho_2)) = \mathbf{n}$.

Substep 4. *Indicating all rootless subtrees of* **D** *to which* ρ_1 *can be mapped by a valuation of* $\mathcal{D}$. We will add an index 1 to the superscript of the label of all nodes **n** for which there exists a valuation f of $\mathcal{D}$ in **D** with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$, or, equivalently, for which there exists a valuation f of $\mathcal{D}'$ in **D** with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$. Now, a valuation f of $\mathcal{D}' - \{\rho_1 \leftarrow (A\rho_2)\}$ can be extended to a valuation of $\mathcal{D}$ in **D** if and only if $\mathbf{par}(f(\rho_2))$ is labeled A. Therefore, in any order and until no further action is possible, we have to perform $\Sigma\pi[A^J \to s,\ A^{J \cup \{1\}}]$ for each $A^J \to s$ with $2 \in J$, $1 \notin J$, and either $A^J \to s$ a production in the current scheme or A^J an attribute in the current scheme and $s = \varepsilon$.

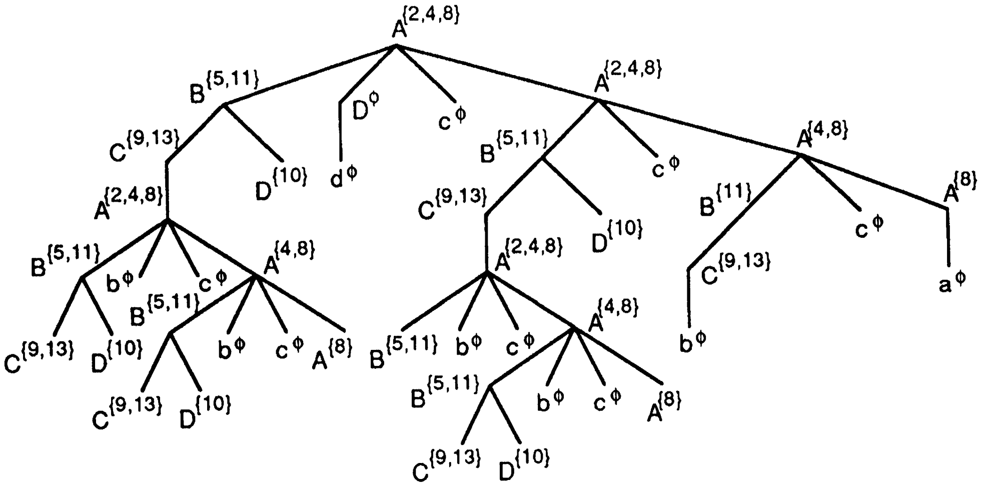


FIG. 26. *Recursively building up all valuations of $\mathcal{D}$ in* **D**.

Substep 5. *Cleaning up.* Using parent substitution, we rename all node labels A^J with $1 \in J$ to $A^{\{1\}}$ and all other node labels B^J with $B \in V$ and $1 \notin J$ to $B^{\emptyset}$.

All nodes **n** labeled A for which there exist a valuation f of $\mathcal{D}$ in **D** with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$ are now indexed by $\{1\}$; all other nodes are indexed by the empty set.

The current instance for our example is now as shown in Fig. 27. There are four valuations of $\mathcal{D}$ in **D**.

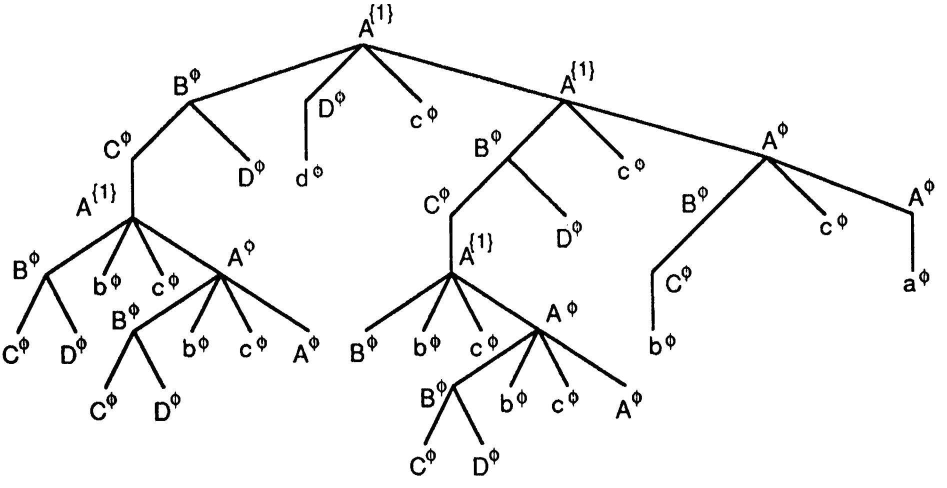


FIG. 27. *Determination of all valuations of $\mathcal{D}$ in* **D**.

Step 3. *Evaluation of all type* 2 *variables under the valuations of $\mathcal{D}$ in* **D**. Once again, let $\rho_{i_1}, \ldots, \rho_{i_m}$ be the type 2 variables in $\mathcal{D}'$ (or, equivalently, in $\mathcal{D}$). We will relabel the nodes in the current instance in such a way that, for each $p = 1, \ldots, m$,

the superscript of the label of a node **n** *contains the index i_p if and only if there exists a valuation f of $\mathcal{D}'$ in* **D** *with* $\mathbf{par}(f(\rho_{i_p})) = \mathbf{n}$ (condition (p')).

Since we will need approximately the same procedure on several other occasions in the following parts of this proof, we will describe it in slightly more general terms than needed right now. Recalling that $\rho_{i_1} = \rho_2$, we can easily satisfy condition (1′) by doing, in any order, until no further action is possible, the parent substitution $\Sigma\pi[B^{\{1\}} \to s, B^{\{1,2\}}]$ for each $B^{\{1\}} \to s$ with either $B^{\{1\}} \to s$ a production in the current scheme or $B^{\{1\}}$ an attribute in the current scheme and $s = \varepsilon$.[13] By an upward iterative procedure, we now enforce conditions (2′)–(m').

Thereto, we perform the following operation for $p := 2$ up to m. Let $\rho_k := \ldots (B\rho_{i_p}) \ldots$ be the declaration in $\mathcal{D}'$ containing ρ_{i_p} in its right-hand side. Then, in any order and until no further action is possible, we do $\Sigma\chi[C^K \to s_1 B^J s_2, B^J, B^{J \cup \{i_p\}}]$ for each production $C^K \to s_1 B^J s_2$ in the current scheme with $C \in V$, $k \in K$, and $i_p \notin J$.

The current instance of our example is now as shown in Fig. 28.

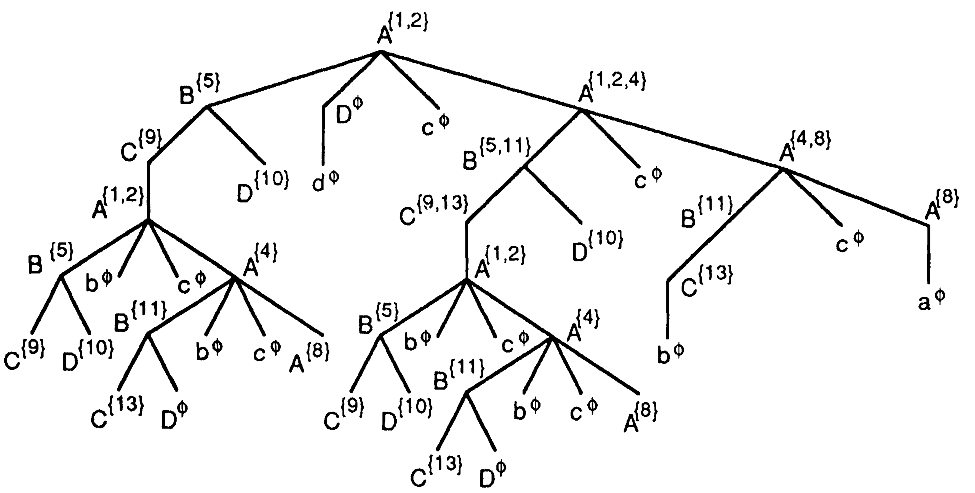


FIG. 28. *Evaluation of all type* 2 *variables under the valuations of* $\mathcal{D}$ *in* **D**.

Step 4. *Evaluation of all type* 1 *variables under the valuations of* $\mathcal{D}$ *in* **D**. We will relabel the nodes in the current instance in such a way that for each type 1 variable ρ_i, the superscript of the label of a node **n** contains the index i if and only if there exists a valuation f of $\mathcal{D}$ in **D** with **top**($f(\rho_i)$) containing **n**. Therefore, we do the following steps:

Substep 1. *Transforming* $\mathcal{D}$. From $\mathcal{D}$ we construct $\mathcal{D}''$ as follows. $\mathcal{D}''$ contains one declaration for each type 1 variable. This type 1 variable is contained in the right-hand side of the declaration. The left-hand side is a type 2 variable. As for $\mathcal{D}'$, the declarations of $\mathcal{D}''$ are obtained by subsequent substitutions.[14] For example, in our example,

$$\mathcal{D}'' = \{\rho_2 := \rho_3 c(A\rho_4),\ \rho_2 := (B\rho_5)\rho_6 c(A\rho_4),\ \rho_4 := \rho_7 c(A\rho_8),$$
$$\rho_4 := (B\rho_{11})\rho_{12} c(A\rho_8),\ \rho_{11} := (C\rho_{13})\rho_{14}\}.$$

Clearly, a valuation of $\mathcal{D}$ in **D** is also a valuation of $\mathcal{D}' \cup \mathcal{D}''$ in **D**; conversely, the extension of a valuation of $\mathcal{D}'$ in **D** to var($\mathcal{D}$) is a valuation of $\mathcal{D}$ in **D** if and only if it is a valuation of $\mathcal{D}' \cup \mathcal{D}''$ in **D**.

[13] Note that, at this stage of the proof, the attribute B will always equal A, the attribute in the declaration $\rho_1 := (A\rho_2)$ for the root of G($\mathcal{D}$).

[14] Again, these substitutions cannot yield illegal terms.

Substep 2. *Relabeling.* In order to enforce for each valuation of $\mathcal{D}$ in $\mathbf{D}$ and for each type 1 variable ρ_i that each node in $\mathbf{top}(f(\rho_i))$ contains the index i in the superscript of its label, we perform the following operations. For later use, they will be once again described in slightly more general terms than required right now. Let $\rho_k := t_1 \ldots t_l$ be the unique declaration in $\mathcal{D}''$ containing ρ_i in its right-hand side. Let in the above declaration t_q, $1 \le q \le l$, be the basic term with $t_q = \rho_i$. Let $\rho_r := \ldots (B\rho_k) \ldots$ be the unique declaration in $\mathcal{D}'$ containing ρ_k in its right-hand side. Then, by step 3 of the construction in this proof, each node $\mathbf{n}$ occurring in $f(\rho_i)$ for some valuation of $\mathcal{D}$ in $\mathbf{D}$ is the child of a node labeled B^J with $k \in J$.

We first relabel all nodes occurring in $f(\rho_i)$ for some valuation of $\mathcal{D}$ in $\mathbf{D}$ labeled by a constant. Thereto, we introduce the following notation. Let $s = \alpha_1 \ldots \alpha_w$ be an arbitrary word over the current attributes and constants. (For $v = 1, \ldots, w$, α_v is an attribute or a constant.) Then we denote by $\bar{s}$ the word $\bar{\alpha}_1 \ldots \bar{\alpha}_w$ with, for $v = 1, \ldots, w$,

$$\bar{\alpha}_v = \begin{cases} a^{K \cup \{i\}} & \text{if } \alpha_v = a^K, a \in T; \\ \alpha_v & \text{otherwise.} \end{cases}$$

The relabeling of the constant nodes is now achieved by performing, in any order and until no further action is possible, the permutation

$$\Pi[B^J \to s_1 \ldots s_{q-1} s_q s_{q+1} \ldots s_l s_{l+1}, s_1 \ldots s_{q-1} \bar{s}_q s_{q+1} \ldots s_l s_{l+1}]$$

for each production $B^J \to s_1 \ldots s_{q-1} s_q s_{q+1} \ldots s_l s_{l+1}$ in the current instance with $k \in J$, in which for $r = 1, \ldots, q-1, q+1, \ldots, l$,

$$s_r = \begin{cases} a^K & \text{if } t_r = a, a \in T; \\ C^K & \text{if } t_r = (C\rho_u), C \in V, \end{cases}$$

for some set K of nonnegative integers, s_{l+1} is a (possibly empty) string of symbols N_u^K,[15] and s_q is an arbitrary word in which the index i does not yet occur as a superscript. Finally, the attribute-labeled nodes occurring in $f(\rho_i)$ for some valuation of $\mathcal{D}$ in $\mathbf{D}$ are labeled by performing, in any order and until no further action is possible, the child substitution $\Sigma\chi[B^J \to s_1 \ldots s_{q-1} s_{q,1} C^L s_{q,2} \ldots s_l, C^L, C^{L \cup \{i\}}]$ for each production $B^J \to s_1 \ldots s_{q-1} s_{q,1} C^L s_{q,2} \ldots s_l$ with $k \in J$ and $i \notin L$ and in which, for $r = 1, \ldots, q-1, q+1, \ldots, l, l+1$, s_r is as above while $s_{q,1}$ and $s_{q,2}$ are arbitrary.

The current instance of our example is now as shown in Fig. 29.

Step 5. *Duplication of rootless subtrees corresponding to variables occurring in equations of* $\mathcal{E}$. We will now take into account the set of equations $\mathcal{E} = \{e_{n+1}, \ldots, e_{n+l}\}$. For each variable ρ_i occurring in an equation of $\mathcal{E}$ and each valuation f of $\mathcal{D}$ in $\mathbf{D}$, we will add to $\mathbf{chln}(\mathbf{n})$, $\mathbf{n}$ being the node with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$, a node $\mathbf{m}$ with $\mathbf{chtrs}(\mathbf{m})$ a "duplicate" of $f(\rho_i)$. Therefore, we do the following steps, keeping in mind that the root ρ_1 of $\mathcal{G}(\mathcal{D})$ does not occur in $\mathcal{E}$:

Substep 1. *Duplicating rootless subtrees corresponding to type* 1 *variables.* For each type 1 variable ρ_i occurring in $\mathcal{E}$, we perform the following procedure. Let $\rho_k := \ldots \rho_i \ldots$ be the unique declaration in $\mathcal{D}''$ containing ρ_i in its right-hand side and let $\rho_r := \ldots (B\rho_k) \ldots$ be the unique declaration in $\mathcal{D}'$ containing the type 2 variable ρ_k in its right-hand side.

In order to be able to apply sidewise duplication on rootless subtrees $f(\rho_i)$, f a valuation of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$, we first have to insert a node $\mathbf{p}$ satisfying $\mathbf{chln}(\mathbf{p}) = \mathbf{top}(f(\rho_i))$. So we do, in any order and until no further action is possible, the node insertion $\mathrm{N}\iota[B^J \to s_1 s_2 s_3, s_1 N_i^{\{0\}} s_3]$ for each $B^J \to s_1 s_2 s_3$ with (i) $k \in J$, (ii) either $B^J \to s_1 s_2 s_3$ a production in the current scheme or B^J an attribute in the current scheme and $s_1 s_2 s_3 = \varepsilon$, and (iii)

[15] Recall that $N_u \notin V$ are some of the auxiliary attributes introduced at the beginning of this proof. At this stage of the proof, however, the information base scheme does not yet contain auxiliary attributes, whence by necessity $s_{l+1} = \varepsilon$.

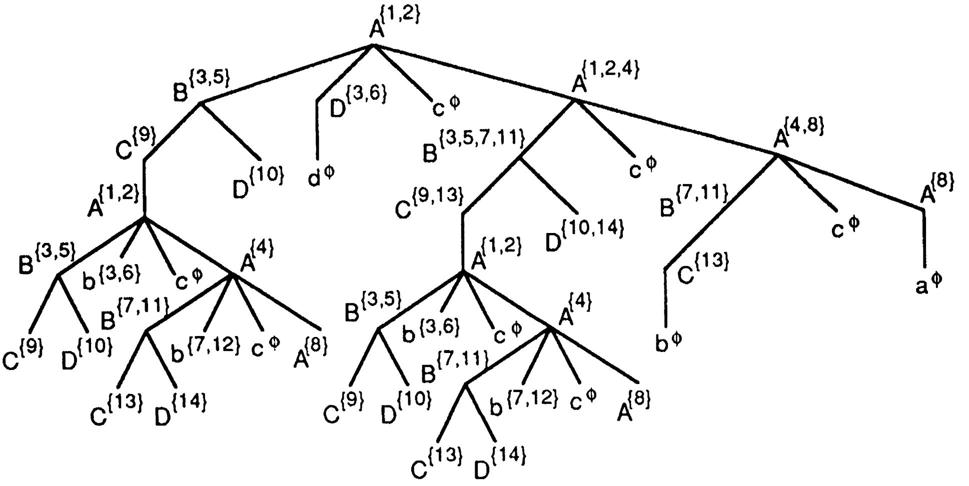


FIG. 29. *Evaluation of all type* 1 *variables under the valuations of* $\mathcal{D}$ *in* **D**.

- for each symbol α^K in s_2, $i \in K$,
- $N_i^{\emptyset}$ not in s_1s_3, and
- for each symbol α^K in s_1s_3,

immediately followed by the sidewise duplication $\Delta\sigma[B^J \rightarrow s_1N_i^{\{0\}}s_2, N_i^{\{0\}}, N_i^{\emptyset}]$ and the node merging $\mathrm{N}\mu[B^J \rightarrow s_1N_i^{\{0\}}s_2N_i^{\emptyset}, N_i^{\{0\}} \rightarrow s_3]$.[16]

Substep 2. *Duplicating rootless subtrees corresponding to type* 2 *variables.* For each type 2 variable ρ_i occurring in $\mathcal{E}$, we perform the following procedure. Let $\rho_k = \ldots(B\rho_i)\ldots$ be the unique declaration in $\mathcal{D}'$ containing ρ_i in its right-hand side.

First, we do, in any order and until no further action is possible, the node insertion $\mathrm{N}\iota[B^J \rightarrow s, N_i^{\{0\}}]$ for each $B^J \rightarrow s$ with $i \in J$, s not containing $N_i^{\emptyset}$, and either $B^J \rightarrow s$ a production in the current scheme or B^J an attribute in the current scheme and $s = \varepsilon$, immediately followed by the sidewise duplication $\Delta\sigma[B^J \rightarrow N_i^{\{0\}}, N_i^{\{0\}}, N_i^{\emptyset}]$ and the node merging $\mathrm{N}\mu[B^J \rightarrow N_i^{\{0\}}N_i^{\emptyset}, N_i^{\{0\}} \rightarrow s]$.

The current instance of our example is now as shown in Fig. 30.

Substep 3. *Eliminating the undesired side effects of the duplication.* For each variable ρ_i occurring in an equation of $\mathcal{E}$ and for each valuation of $\mathcal{D}$ in **D**, it is now our intention to propagate upward the "duplicate" of $f(\rho_i)$, the parent of which is labeled $N_i^{\emptyset}$, until that parent node becomes a child of the node **n** with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$. Unfortunately, we may run into trouble if the duplicate of $f(\rho_i)$ contains a "copy of the duplicate" of $g(\rho_k)$, g another valuation of $\mathcal{D}$ in **D** and ρ_k an arbitrary variable, *without containing a corresponding "copy" of* $g(\rho_1)$. Such copies of duplicates will be called *undesired copies of duplicates.*

For example, in our example, "copies of duplicates" occur. For instance, if f is the valuation of $\mathcal{D}$ in **D** for which $f(\rho_1) = (\mathbf{D})$ and g is the valuation of $\mathcal{D}$ in **D** for which $g(\rho_1)$ is the leftmost subtree of **D** of which the root is labeled A (see Fig. 29), then the rightmost occurrence of Δ_1 in Fig. 30 is the duplicate of $f(\rho_{13})$. Clearly, the leftmost occurrence of Δ_1 in Fig. 30 contains $g(\rho_9)$ (which is empty) as well as the duplicate of $g(\rho_9)$. Hence the

[16] Observe that if $f(\rho_i)$ is the empty rootless subtree, then, by the first operation, one $N_i^{\{0\}}$-labeled leaf node is inserted in an arbitrary place between the children of the corresponding B^J-node, not necessarily corresponding with the place of ρ_i in the right-hand side of the declaration $\rho_k := \ldots\rho_i\ldots$ in $\mathcal{D}''$. Fortunately, this nondeterminism gets immediately eliminated by the subsequent sidewise duplication and node merging.

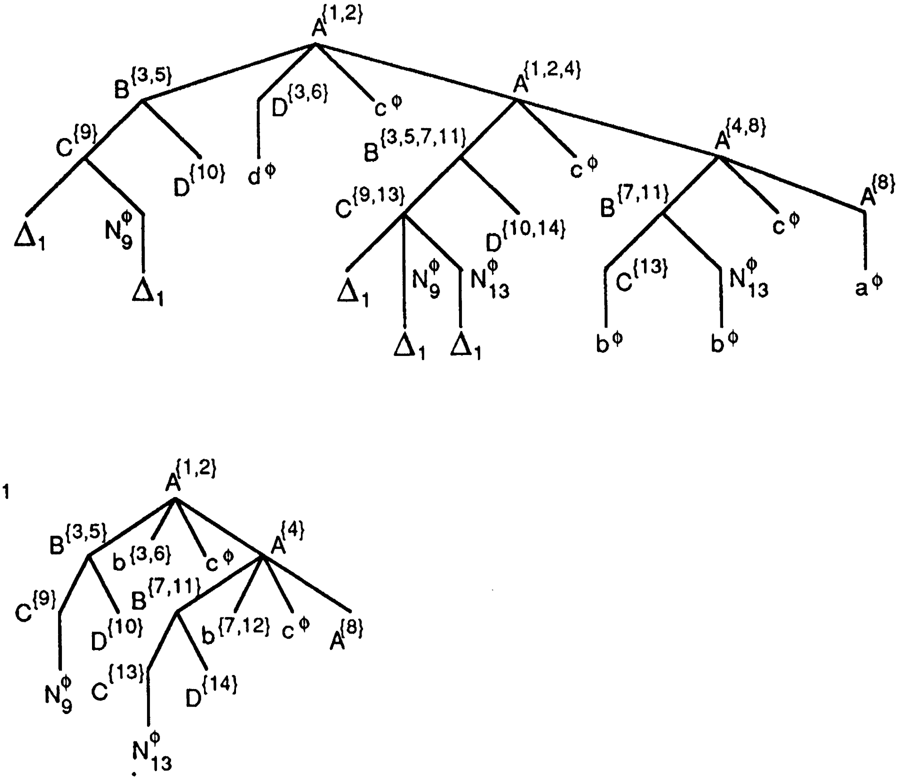


FIG. 30. *Duplication of rootless subtrees corresponding to type* 1 *and type* 2 *variables.*

duplicate of $f(\rho_{13})$ contains a copy of the duplicate of $g(\rho_9)$. However, the duplicate of $f(\rho_{13})$ also contains a copy of $g(\rho_1)$ (namely the duplicate of $f(\rho_{13})$ itself). Such copies of duplicates do not cause problems later; in fact, we will need them in order to generate the correct result. Undesired copies of duplicates do not occur in our example, and, as a consequence, the operations described below will not alter the instance in Fig. 30. An example in which undesired copies of duplicates do occur is given in the Appendix. There, it is also shown why these undesired copies of duplicates cause trouble.

We now remove undesired copies of duplicates as follows.[17] First, we remove all indices except the index 1 from the superscripts of the node labels using the cleaning-up procedure in step 2.5 of this construction. Then, we reintroduce the other indices by repeating steps 3 and 4. The way the auxiliary attributes N_u^K are treated there prevents the indexing to be propagated downward through the parent nodes of duplicates.

Next, we will add an index 0 to the N_u^K corresponding to undesired copies of duplicates.[18] They are recognized as follows. For each type 1 variable ρ_i occurring in $\mathcal{E}$, we do the child substitution $\Sigma\chi[B^J \to s_1 N_u^K s_2, N_u^K, N_u^{K\cup\{0\}}]$ for all productions $B^J \to s_1 N_u^K s_2$ in the current

[17] Of course, *desired* copies of duplicates can in turn contain *undesired* "copies of copies of duplicates," and so on, down to arbitrarily deep levels in the information base instance. While we choose not to complicate this already involved proof any further by explaining the problem only at the highest level where it can occur, the procedure described here is nevertheless general and removes undesired copies at any level in the information base instance.

[18] Note that, even if step 5.3 is initiated with an instance in which the superscripts of auxiliary attributes are arbitrary finite sets of nonnegative integers, we always have at this point that either $K = \{1\}$ or $K = \emptyset$, so K never contains 0.

scheme for which the superscript of no symbol in s_1 contains the index i. Similarly, for each type 2 variable ρ_i occurring in $\mathcal{E}$, we do the child substitution $\Sigma\chi[B^J \to s_1 N_u^K s_2,\ N_u^K,\ N_u^{K\cup\{0\}}]$ for all productions $B^J \to s_1 N_u^K s_2$ in the current scheme with $i \notin J$. Finally, undesired copies of duplicates are now removed by the node deletions $N\delta[N_u^K]$ with N_u^k in the current scheme, N_u an auxiliary attribute, and $0 \in K$.

If for some type 1 or type 2 variables ρ_i and ρ_k and for some valuation f of $\mathcal{D}$ in $\mathbf{D}$, $f(\rho_i)$ and $f(\rho_k)$ are isomorphic, then all their respective duplicates are also isomorphic, by Theorem 3.13. In general, some of these duplicates will have disappeared and some others will have been "trimmed" after step 5.3. However, the operations in step 5.3 preserve isomorphism.

Substep 4. *Propagating duplicates upward in the instance.* For each variable ρ_i occurring in an equation of $\mathcal{E}$ and for each valuation of $\mathcal{D}$ in $\mathbf{D}$, we will now move upward the "duplicate" of $f(\rho_i)$, the parent of which is labeled $N_i^{\emptyset}$, until that parent node becomes a child of the node $\mathbf{n}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$. This is achieved by performing, in any order and until no further action is possible, the upward duplication $\Delta\upsilon[C^K \to s_1 D^L s_2,\ D^L \to s_3 N_k^{\emptyset} s_4,\ N_k^{\emptyset},\ N_{k+n}^{\emptyset}]$, immediately followed by the node deletion $\mathrm{N}\delta[N_k^{\emptyset}]$, and this for all productions $C^K \to s_1 D^L s_2$ and $D^L \to s_3 N_k^{\emptyset} s_4$ in the current scheme with $k \operatorname{div} n + 2 \leq \mathrm{d}(\rho_{k \bmod n})$. As a consequence of this condition, duplicates of variables of depth 1 in $\mathcal{G}(\mathcal{D})$ are not moved upward; indeed, they already are in their right position.

Finally, in order to be able to immediately recognize the variable to which an $N_k^{\emptyset}$-labeled node corresponds, we perform, in any order and until no further action is possible, the child substitution $\Sigma\chi[A^J \to s_1 N_k^{\emptyset} s_2,\ N_k^{\emptyset},\ N_{k \bmod n}^{\emptyset}]$ for each production $A^J \to s_1 N_k^{\emptyset} s_2$ in the current scheme with $k > n$.

For each variable ρ_i occurring in an equation of $\mathcal{E}$ and for each valuation of $\mathcal{D}$ in $\mathbf{D}$ the A-labeled node $\mathbf{n}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$ has a child $\mathbf{m}$ labeled $N_i^{\emptyset}$ with $\mathbf{chtrs}(\mathbf{m})$ a "duplicate" of $f(\rho_i)$. Because of Theorem 3.13, it suffices to compare these "duplicates" in order to decide the isomorphism of the original $f(\rho_i)$'s.

The current instance of our example is now as shown in Fig. 31.

Observe that the $N_i^{\emptyset}$-labeled nodes always occur to the right of each of their siblings labeled by another symbol.

Step 6. *Determination of all valuations of $\mathcal{D}$ and $\mathcal{E}$ in* $\mathbf{D}$. We will relabel by $A^{\{1\}}$ all nodes $\mathbf{n}$ for which there exists a valuation f of $\mathcal{D}$ and $\mathcal{E}$ in $\mathbf{D}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$. (The superscript of the label of all other nodes will be replaced by the empty set.) Therefore, we do the following steps:

Substep 1. *Indicating the equations of $\mathcal{E}$ satisfied by the valuations of $\mathcal{D}$ in* $\mathbf{D}$. For each equation $e_k \equiv \rho_{k_1} = \rho_{k_2}$ in $\mathcal{E}$, $n+1 \leq k \leq n+l$ and for each valuation f of $\mathcal{D}$ in $\mathbf{D}$, we will add the index k to the superscript of the label of the node $\mathbf{n}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$ if and only if f satisfies e_k, i.e., if $f(\rho_{k_1})$ and $f(\rho_{k_2})$ are isomorphic. This is achieved by performing, in any order and until no further action is possible, the parent equality substitution $\Sigma\sigma[A^J \to s,\ N_{k_1}^{\emptyset},\ N_{k_2}^{\emptyset},\ A^{J\cup\{k\}}]$ for all productions $A^J \to s$ in the current scheme with $N_{k_1}^{\emptyset}$ and $N_{k_2}^{\emptyset}$ in s and $k \notin J$.

In our example, we only have one equation: $e_{15} \equiv \rho_9 = \rho_{13}$. Three of the four valuations of $\mathcal{D}$ in $\mathbf{D}$ satisfy this equation. Their corresponding A-nodes see an index 15 added to the superscript of their label. The current instance in our example is now shown in Fig. 32.

Substep 2. *Relabeling and cleaning up.* In any order and until no further action is possible, we perform $\Sigma\pi[A^J \to s,\ A^{\{1\}}]$ for each production $A^J \to s$ in the current scheme with $\{n+1, \ldots, n+l\} \subseteq J$.

Using parent substitution, we now rename all attribute node labels B^J with $B \in V$ and $1 \notin J$ to $B^{\emptyset}$; using permutation, we rename all constant node labels a^J with $a \in T$ by $a^{\emptyset}$. Finally, we remove all $N_i^{\emptyset}$, as well as the subtrees of which they are the root, using node deletion.

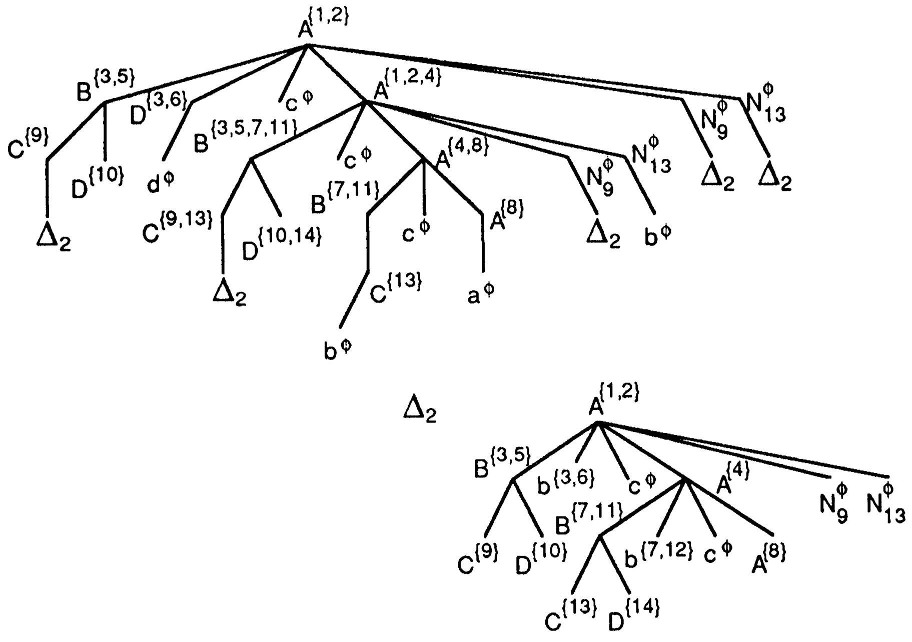


FIG. 31. *Duplication of rootless subtrees corresponding to variables occurring in equations of $\mathcal{E}$.*

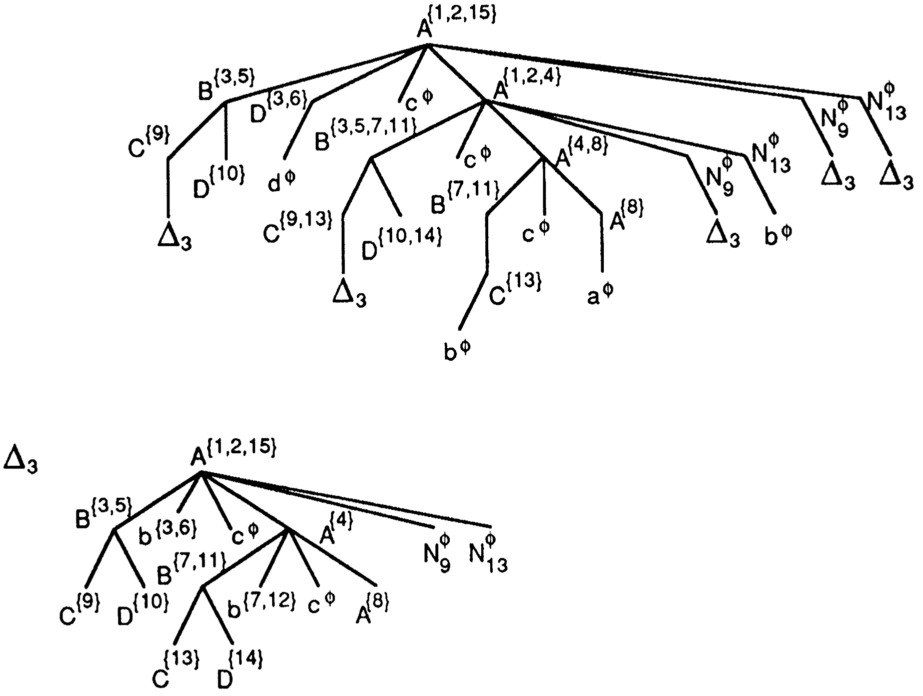


FIG. 32. *The result of marking the equations of $\mathcal{E}$ satisfied by the valuations of $\mathcal{D}$ in* **D**.

Now, all nodes **n** for which there exists a valuation f of $\mathcal{D} \cup \mathcal{E}$ in **D** with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$ are labeled $A^{\{1\}}$; all other nodes are labeled with an attribute or a constant indexed by the empty set.

Comparing the instances obtained at the end of step 2 and this step respectively, we have now eliminated all valuations of $\mathcal{D}$ in **D** that are not valuations of $\mathcal{D}$ and $\mathcal{E}$ in **D**. In our example, one of the valuations identified in Fig. 27, namely that for which the corresponding A-node did not receive an index 15 in Fig. 32, is eliminated.

Step 7. *Evaluation of all variables under the valuations of* $\mathcal{D}$ *and* $\mathcal{E}$ *in* **D**. We will relabel the nodes in the current instance in such a way that

- for each type 2 variable ρ_i, the superscript of the label of a node **n** contains the index i if and only if there exists a valuation f of $\mathcal{D}$ and $\mathcal{E}$ in **D** with $\mathbf{par}(f(\rho_i)) = \mathbf{n}$;
- for each type 1 variable ρ_i, the superscript of the label of a node **n** contains the index i if and only if there exists a valuation f of $\mathcal{D}$ and $\mathcal{E}$ in **D** with $\mathbf{top}(f(\rho_i))$ containing **n**.

Thereto, it suffices to repeat steps 3 and 4 of the construction of this proof, starting from the current instance. In our example, the resulting instance is then as shown in Fig. 33.

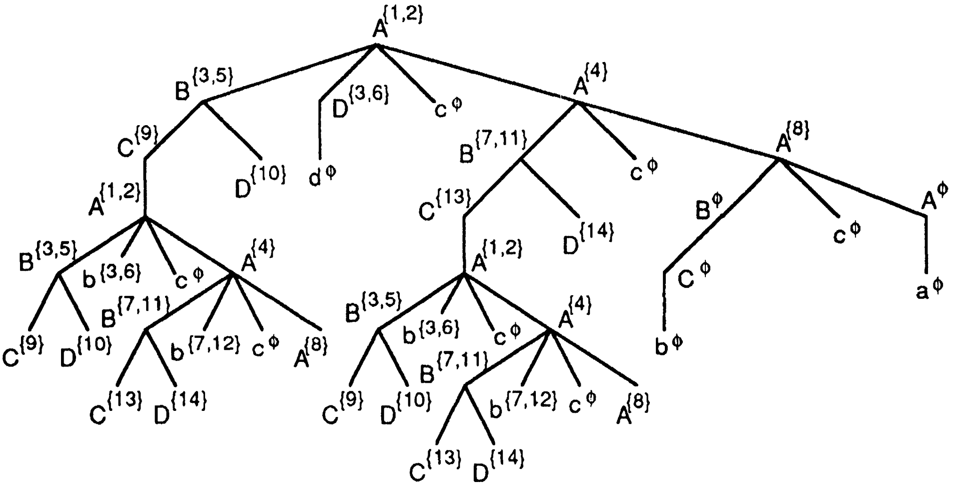


FIG. 33. *Evaluation of all variables under the valuations of* $\mathcal{D}$ *and* $\mathcal{E}$ *in* **D**.

Step 8. *Duplication of rootless subtrees corresponding to variables in u.* Now that all valuations of $\mathcal{D}$ and $\mathcal{E}$ in **D** have been fully specified, we can start with the simulation of the actual transformation. Thereto, for each variable ρ_i in the substitution term u and each valuation f of $\mathcal{D}$ and $\mathcal{E}$ in **D**, we will add to $\mathbf{chln}(\mathbf{n})$, **n** being the node with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$, a node **m** with $\mathbf{chtrs}(\mathbf{m})$ a "duplicate" of $f(\rho_i)$. Therefore, we do the following steps, keeping in mind that ρ_1 does not occur in u:

Substep 1. *Duplicating rootless subtrees corresponding to type* 1 *variables.* Thereto, we simply repeat step 5.1 of the construction in this proof.

Substep 2. *Duplicating rootless subtrees corresponding to type* 2 *variables.* In order to duplicate type 2 variables, we *cannot* simply repeat step 5.2. This is because of the different ways in which type 1 and type 2 variables are treated in Definition 4.18. Indeed, if ρ_i is a variable of type 1 in u, f is a valuation of $\mathcal{D}$ and $\mathcal{E}$ in **D** with $f(\rho_i) = (\mathbf{D}_1, \dots, \mathbf{D}_w)$, and g is an E-transformation, then we have to compute $g((\mathbf{D}_1), \mathbf{D}_1) \dots g((\mathbf{D}_m), \mathbf{D}_m)$. If, on the other hand, ρ_i is a type 2 variable, we have to compute $g(f(\rho_i), \mathbf{p}f(\rho_i))$, where $\mathbf{p} = \mathbf{par}(f(\rho_i))$.

Therefore, if ρ_i is a type 2 variable in u and f is a valuation of $\mathcal{D}$ and $\mathcal{E}$ in **D**, we will duplicate the *completion* of $f(\rho_i)$ with its parent node, rather than $f(\rho_i)$ itself. Thereto, we perform the following procedure for each type 2 variable ρ_i in u. Let $\rho_k = \ldots(B\rho_i)\ldots$ be the unique declaration in $\mathcal{D}'$ containing ρ_i in its right-hand side.

First, we do, in any order and until no further action is possible, the node insertion N$\iota[B^J \to s, N_i^{J\cup\{0\}}]$ for each $B^J \to s$ with $i \in J$, s not containing N_i^J, and either $B^J \to s$ a production in the current scheme or B^J an attribute in the current scheme and $s = \varepsilon$, immediately followed by the sidewise duplication $\Delta\sigma[B^J \to N_i^{J\cup\{0\}}, N_i^{J\cup\{0\}}, N_i^J]$ and the node merging N$\mu[B^J \to N_i^{J\cup\{0\}} N_i^J, N_i^{J\cup\{0\}} \to s]$.

In comparison with step 5.2, we now also duplicated the superscript of the label of the parent node of $f(\rho_i)$.

Substep 3. *Eliminating the undesired side effects of the duplication.* We repeat step 5.3 of the construction in this proof.

Substep 4. *Propagating duplicates upward in the instance.* We repeat step 5.4 of the construction in this proof.

For each variable ρ_i in the substitution term u and for each valuation of $\mathcal{D}$ in **D**, the A-labeled node **n** with **top**$(f(\rho_1)) = (\mathbf{n})$ has a child **m** labeled N_i^J for some $J \subseteq \{1, \ldots, n\}$ with **chtrs**(**m**) a "duplicate" of $f(\rho_i)$.

The current instance of our example is now as shown in Fig. 34.

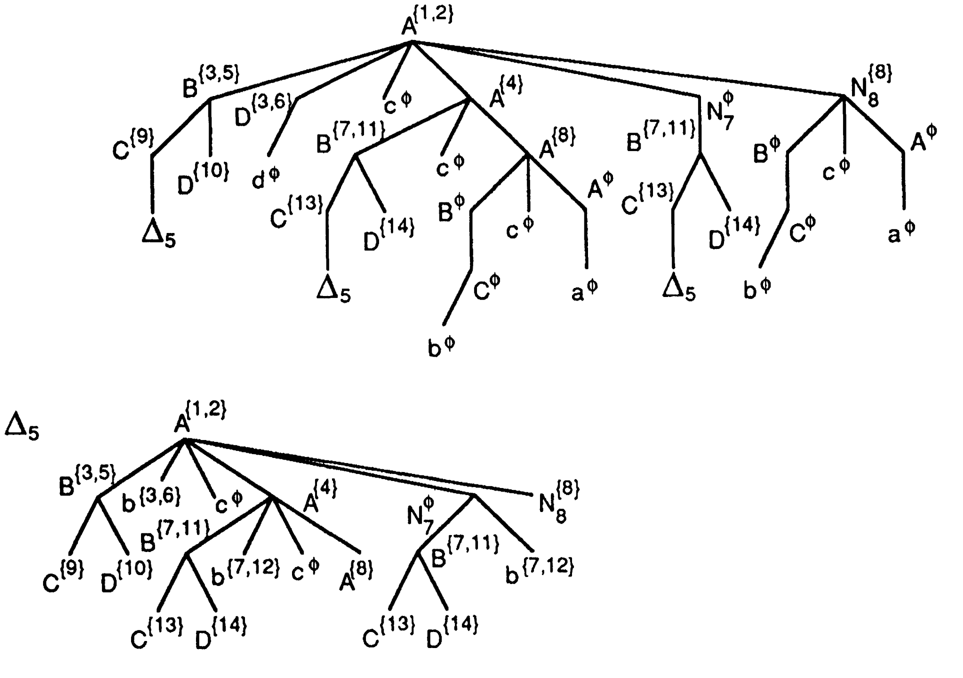


FIG. 34. *Duplication of rootless subtrees corresponding to variables in u.*

Observe that the N_i^J-labeled nodes always occur to the right of each of their siblings labeled by another symbol.

Step 9. *Downward propagation of duplicates to the place where they have to be inserted.* If ρ_j, the left-hand side of the substitution clause $\rho_j \leftarrow u$, equals ρ_1, the root of $\mathcal{G}(\mathcal{D})$, then no alterations are made.

Otherwise, for each variable ρ_i in u and for each valuation of $\mathcal{D}$ and $\mathcal{E}$ in **D**, we will move downward the "duplicate" of $f(\rho_i)$ until it becomes a "sibling" of $f(\rho_j)$. Therefore, we do the following steps:

Substep 1. *Identifying the paths for the downward propagation of duplicates.* We will identify these paths with the sequence of variables $\rho_{q_1}, \ldots, \rho_{q_{d(\rho j)}}$ where, for $1 \leq l \leq \mathrm{d}(\rho_j)$, ρ_{q_l} is the unique ancestor of ρ_j in $\mathcal{G}(\mathcal{D})$ that is of type 2 and has depth l.

Remembering that $\rho_1 := (A\rho_2)$ is the declaration of the root of $\mathcal{G}(\mathcal{D})$, we always have $\rho_{q_1} = \rho_2$. In our example, we have $\rho_j = \rho_5$, $\mathrm{d}(\rho_5) = 2$, and $\rho_{q_1} = \rho_2$, $\rho_{q_2} = \rho_5$.

Substep 2. *Propagating duplicates downward in the instance.* For each variable ρ_i in u and for each valuation of $\mathcal{D}$ and $\mathcal{E}$ in **D**, we will now move downward the "duplicate" of $f(\rho_i)$ along the path identified in the previous step, until it becomes a "sibling" of $f(\rho_j)$, i.e., until it has the same parent node as $f(\rho_j)$, or, equivalently, until it has the same parent node as $f(\rho_{q_{d(\rho_j)}})$.

This is achieved by performing, in any order and until no further action is possible, the downward duplication $\Delta\delta[B^K \to s_1 C^L s_2 N_k^J s_3, C^L \to s_4, N_k^J, N_{k+n}^J]$, immediately followed by the node deletion $\mathrm{N}\delta[N_k^J]$, for each production $B^K \to s_1 C^L s_2 N_k^J s_3$ in the current scheme with $C \in V$, $k \operatorname{div} n + 2 \leq \mathrm{d}(\rho_j)$, and $q_{k \operatorname{div} n+2} \in L$ (whence $q_{k \operatorname{div} n+1} \in K$), and for each $C^L \to s_4$ with either $C^L \to s_4$ a production in the current scheme or $s_4 = \varepsilon$.

Finally, in order to be able to immediately recognize the variable to which an $N_k^{\emptyset}$-labeled node corresponds, we perform, in any order and until no further action is possible, the child substitution $\Sigma\chi[B^K \to s_1 N_k^J s_2, N_{k \bmod n}^J]$ for each production $B^K \to s_1 N_k^J s_2]$ with $q_{d(\rho_j)} \in K$ and $k > n$.

The current instance of our example is now as shown in Fig. 35.

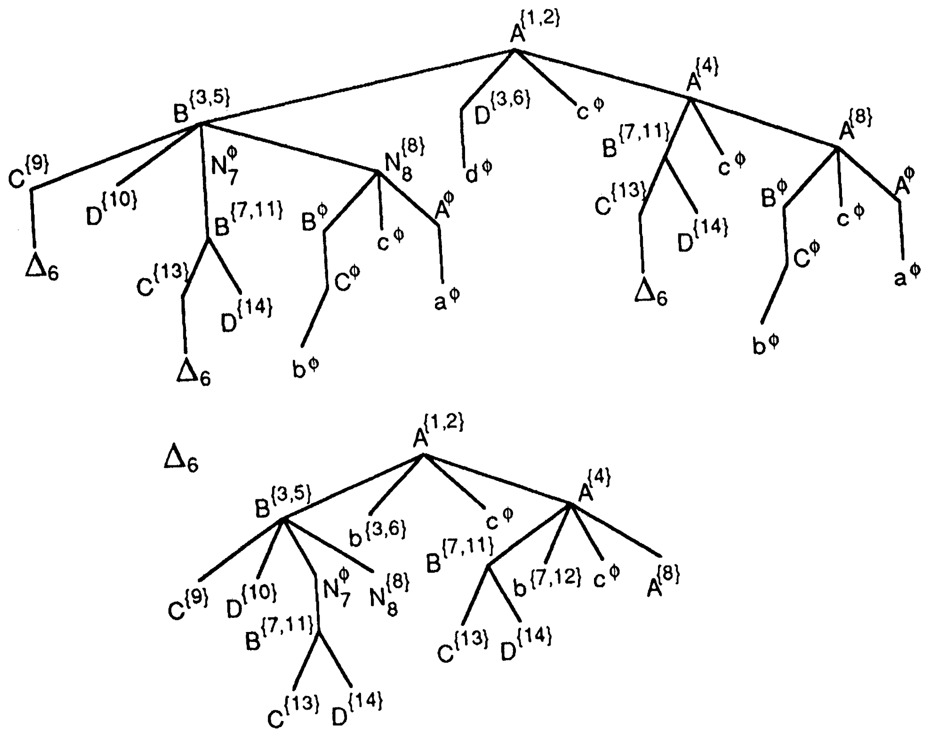


FIG. 35. *Downward propagation of duplicates to the place where they have to be inserted.*

Observe that step 9.2 does not produce alterations if $l = 1$, i.e., if $\rho_j \equiv \rho_2$. Indeed, in that case, all duplicates already have the same parent node as $f(\rho_j) \equiv f(\rho_2)$, namely the A-labeled node $\mathbf{n}$ with $\mathbf{top}(f(\rho_1)) = (\mathbf{n})$.

Step 10. *The actual transformation in the case when in the substitution clause $\rho_j \leftarrow u$, $\rho_j \equiv \rho_1$.* We distinguish two cases.

Case 1. *The substitution clause is $\rho_1 \leftarrow \varepsilon$.* Then, in any order and until no further action is possible, we do the node deletion $\mathrm{N}\delta[A^J]$ for each attribute in the current scheme with $1 \in J$. Finally, we remove the superscripts from all labels using parent substitution (for the attribute nodes) and permutation (for the constant nodes).

The resulting instance is $[\rho_1 \leftarrow \varepsilon \mid \mathcal{D} \cup \mathcal{E}](\mathbf{D})$.

Case 2. *The substitution clause has the form $\rho_1 \leftarrow (B\rho_i)$ with $B \in V$ and $\rho_i \in$* var$(\mathcal{D})$. First, we mark the subtrees to be deleted with the index $n+1$ by doing, in any order and until no further action is possible, the child substitution

$$\Sigma\chi[C^J \to s_1 D^K s_2, D^K, D^{K \cup \{n+1\}}]$$

for each production $C^J \to s_1 D^K s_2$ in the current scheme with $1 \in J$ (whence $C = A$ or $C = N_i$), $n+1 \notin K$, and $D \neq N_i$. Then, we actually remove these subtrees using the node deletions $\mathrm{N}\delta[D^K]$ for all attributes D in the current scheme with $n+1 \in K$. Next, we remove the auxiliary attributes by applying, in any order and until no further action is possible, the node merging $\mathrm{N}\mu[A^J \to N_i^K, N_i^K \to s]$ for all productions $A^J \to N_i^K$ and $N_i^K \to s$ in the current scheme with $1 \in J$. Finally, in any order and until no further action is possible, we perform the parent substitution $\Sigma\pi[A^J \to s, B]$ for each $A^J \to s$ with $1 \in J$ and either $A^J \to s$ a production in the current scheme or A^J an attribute in the current scheme and $s = \varepsilon$. Using parent substitution (for the attribute nodes) and permutation (for the constant nodes), we remove the index sets from all other labels.

The resulting instance is $[\rho_1 \leftarrow (B\rho_i) \mid \mathcal{D} \cup \mathcal{E}](\mathbf{D})$.

Step 11. *The actual transformation in the case when in $\rho_j \leftarrow u$, ρ_j is of type* 1 *in $\mathcal{D}$.* Let $k = q_{d(\rho_j)}$ and let $\rho_k := t_1 \ldots t_l$ be the unique declaration in $\mathcal{D}''$ containing ρ_i in its right-hand side. Let in the above declaration t_q, $1 \leq q \leq l$, be the basic term with $t_q = \rho_j$. Let $\rho_r := \ldots (B\rho_k) \ldots$ be the unique declaration in $\mathcal{D}'$ containing ρ_k in its right-hand side. Let $u = u_1 \ldots u_w$ with $u_1, \ldots, u_w$ basic terms. We then do the following steps:

Substep 1. *Identifying and removing subtrees to be deleted.* In any order and until no further action is possible, we do the node insertion $N\iota[C^J \to s_1 s_2 s_3, s_1 N_0^{\emptyset} s_3]$ for each production $C^J \to s_1 s_2 s_3$ in the current scheme with $k \in J$, s_2 the substring of $s_1 s_2 s_3$ of *all* symbols containing the index j in the superscript of their label, and $s_2 \neq \varepsilon$, immediately followed by the node deletion $N\delta[N_0^{\emptyset}]$.[19]

Substep 2. *Rearranging subtrees to be substituted.* In any order and until no further action is possible, we perform the permutation $\Pi[C^J \to s_1 s_2 s_3, s_1 s_4 s_2]$ for each $C^J \to s_1 s_2 s_3$ with (i) $k \in J$, (ii) either $C^J \to s_1 s_2 s_3$ a production in the current scheme, or C^J an attribute in the current scheme and $s_1 s_2 s_3 = \varepsilon$, and (iii)

- the superscript of no symbol in $s_1 s_2 s_3$ contains the index j;
- s_1 consists of $q-1$ symbols;
- s_2 consists of $l-q$ symbols;
- s_3 only consists of symbols N_i^K with N_i an auxiliary attribute; and
- $s_4 = \alpha_1 \ldots \alpha_l$ with, for $v = 1, \ldots, l$,

$$\alpha_v = \begin{cases} a & \text{if } u_v = a; \\ N_i^K & \text{if } u_v = \rho_i \text{ and } N_i^K \text{ is in } s_3; \\ N_i^K & \text{if } u_v = (D\rho_i) \text{ and } N_i^K \text{ is in } s_3. \end{cases}$$

(Observe that s_4 is uniquely defined by the above conditions.)

[19] Observe that, in the above production, either $C = B$ (B being the attribute in the declaration $\rho_r := \ldots (B\rho_k) \ldots$) or C is an auxiliary attribute.

Substep 3. *Cleaning up.* For each indexed attribute N_i^K in the current scheme with ρ_i a variable that occurs in u as a basic term of type 2, say $(D\rho_i)$, we do the parent substitution $\Sigma\pi[N_i^K \to s, D]$ for each production $N_i^K \to s$ in the current scheme. Then, for each indexed attribute N_i^K in the current scheme with ρ_i a variable that occurs in u as a basic term of type 1, we do the node merging $N\mu[B^J \to s_1 N_i^K s_2, N_i^K \to s_3]$ for all productions $B^J \to s_1 N_i^K s_2$ and $N_i^K \to s_3$ in the current scheme with $k \in J$ (B being the attribute in the declaration $\rho_r := \dots (B\rho_k) \dots$). Finally, we remove all remaining index sets from node labels using parent substitution (for the attribute nodes) and permutation (for the constant nodes).

The resulting instance is $[\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$.

Step 12. *The actual transformation in the case that in $\rho_j \leftarrow u$, ρ_j is of type* 2 *in $\mathcal{D}$.* If ρ_j is of type 2 in $\mathcal{D}$, then $q_{d_{(\rho_j)}} = j$. Let $\rho_r := \dots (B\rho_j) \dots$ be the unique declaration in $\mathcal{D}'$ containing ρ_j in its right-hand side. Let $u = u_1 \dots u_w$ with $u_1, \dots, u_w$ basic terms. We then do the following steps:

Substep 1. *Identifying and removing subtrees to be deleted.* In any order and until no further action is possible, we do the node insertion $N\iota[C^J \to s_1 s_2, N_0^{\emptyset} s_2]$ for each production $C^J \to s_1 s_2$ in the current scheme with $j \in J$, s_2 the substring of $s_1 s_2$ of *all* auxiliary attributes, and $s_1 \neq \varepsilon$, immediately followed by the node deletion $N\delta[N_0^{\emptyset}]$.[20]

Substep 2. *Rearranging subtrees to be substituted.* In any order and until no further action is possible, we perform the permutation $\Pi[C^J \to s_1, s_2]$ for each $C^J \to s_1$ with $j \in J$, either $C^J \to s_1$ a production in the current scheme, or C^J an attribute in the current scheme and $s_1 = \varepsilon$, and $s_2 = \alpha_1 \dots \alpha_l$ with, for $v = 1, \dots, l$,

$$\alpha_v = \begin{cases} a & \text{if } u_v = a; \\ N_i^K & \text{if } u_v = \rho_i \text{ and } N_i^K \text{ is in } s_1; \\ N_i^K & \text{if } u_v = (D\rho_i) \text{ and } N_i^K \text{ is in } s_1. \end{cases}$$

(Observe that s_1 only consists of auxiliary attributes and that s_2 is uniquely defined by the above conditions.)

Substep 3. *Cleaning up.* We repeat step 11.3 in the construction of this proof.

The resulting instance is $[\rho_j \leftarrow u \mid \mathcal{D} \cup \mathcal{E}]$. Particularly, in our example, the operations above finally yield the information base instance shown in Fig. 19.

Finally note that all constructions in this proof were done at scheme level, i.e., they do not depend on the instance under consideration. □

Hence the grammatical algebra and the grammatical calculus are equivalent. As Codd concluded for the relational model, this equivalence gives a naturalness to both languages. However, it still requires further investigation to find a precise language-independent characterization for the expressive power of the grammatical algebra and calculus.

7. Conclusions and future work. In this paper a simple model for representing the hierarchical structure in information is proposed. Two methods for querying in this data model are given and shown to be equivalent. The expressive power of these querying methods is not yet clear, however. In particular, it is not known how these methods are related to querying facilities in other data models, that can be simulated by the grammatical model. Furthermore, we are looking for a well-adapted interface that is integrated in a more general environment. It is remarkable that there seems to be no fundamental distinction between updating and querying in this model. Other aspects, such as transforming several given trees into one result tree, constraint checking, and implementation strategies, are under investigation.

Although it can be considered as an extension of the relational model, the grammatical model, because it is hierarchical in nature, is of course not suited for all database applications.

[20] Observe that, in the above production, either $C = B$ (B being the attribute in the declaration $\rho_r := \dots (B\rho_j) \dots$) or C is an auxiliary attribute.

In particular, the notion of "shared component" is difficult to express in a tree. It is therefore interesting to look for a characterization of the semantic expressiveness of the grammatical model. On the other hand, one could also look for "network-like" extensions of this model, using the theory of graph-grammars (e.g., [15]).

Appendix. As promised in step 5.3 of the construction in the proof of Theorem 6.1, we shall now exhibit an example in which duplication yields undesired side effects.

Example. Consider the information base instance **D** of Fig. 36 (over some appropriate scheme), let

$$\mathcal{D} = \{\rho_1 := (A\rho_2),\ \rho_2 := (A\rho_3)(B\rho_4)(C\rho_5),\ \rho_3 := (A\rho_6)(B\rho_7)(C\rho_8)\},$$

and let E be an arbitrary calculus expression involving $\mathcal{D}$ and the set of equations $\mathcal{E} = \{\rho_6 = \rho_8\}$.

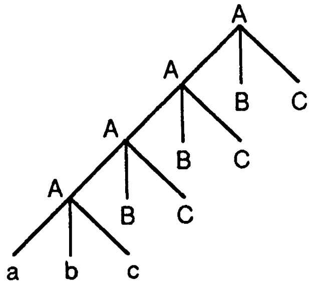


FIG. 36. *The information base instance* **D**.

If we apply the construction in the proof of Theorem 6.1 to the instance **D** up to step 4, i.e., until all valuations of $\mathcal{D}$ are determined and fully specified, we obtain the instance in Fig. 37.

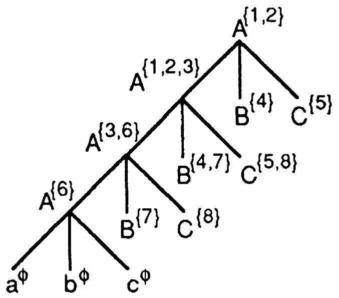


FIG. 37. *The instance* **D** *after determination and full specification of all valuations of* $\mathcal{D}$ *in* **D**.

Clearly, there are two valuations of $\mathcal{D}$ in **D**. One of this, say f, is determined by $f(\rho_1) = (\mathbf{D})$. For the other one, say g, $g(\rho_1)$ is the leftmost component of **chtrs(rt(D))**. Since ρ_6 and ρ_8 are the only variables occurring in an equation of $\mathcal{E}$, the only subtrees to be duplicated are $f(\rho_6)$, $f(\rho_8)$, $g(\rho_6)$, and $g(\rho_8)$. The result of applying steps 5.1 and 5.2 to the instance of Fig. 37 is shown in Fig. 38.

Now observe that the duplicate of $f(\rho_6)$ contains a "copy" of the duplicate of $g(\rho_6)$ as well as a "copy" of the duplicate of $g(\rho_8)$. Clearly, the duplicate of $f(\rho_6)$ does not contain a duplicate of $g(\rho_1)$.

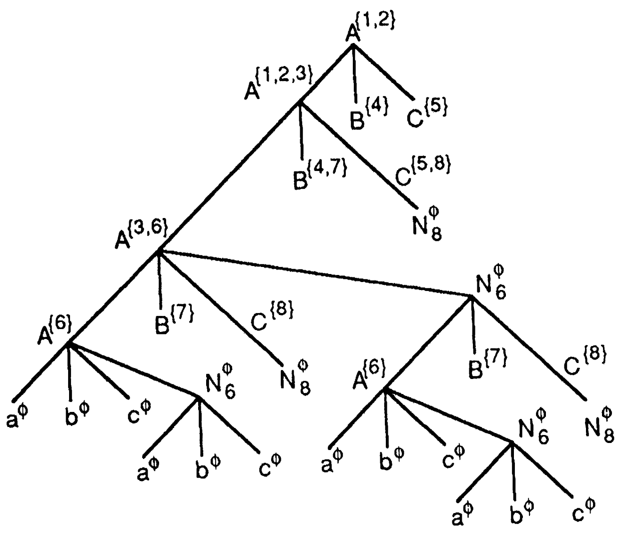


FIG. 38. *The instance of Fig. 37 after duplication of $f(\rho_6)$, $f(\rho_8)$, $g(\rho_6)$, and $g(\rho_8)$.*

If we now would try to apply step 5.4 of the construction in the proof of Theorem 6.1 straight away, we would have to move all duplicates (and hence also all copies of duplicates) two levels upward. We leave it to the reader to verify that, for ρ_6, this would result in two nodes at the same level (more concrete, as children of the root) with the same attribute label $N_{24}^{\emptyset}$, which would imply that the result is undefined, and this is obviously not what we want. Clearly, a duplicate of $g(\rho_6)$ does not belong at that level. Luckily, the relabeling procedure of step 5.3 reevaluates the valuations of $\mathcal{D}$ in **D** and prevents them from being propagated downward through nodes labeled by auxiliary attributes. The result of the relabeling is shown in Fig. 39.

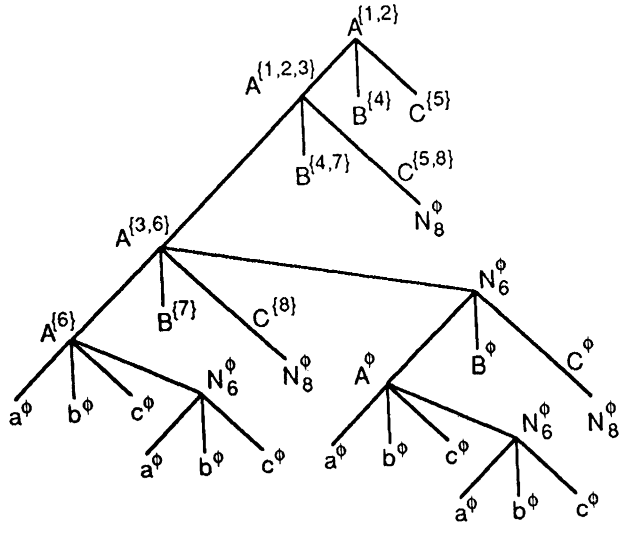


FIG. 39. *The instance of Fig. 38 after relabeling.*

The undesired node with label $N_6^{\emptyset}$ (as well as the undesired node with label $N_8^{\emptyset}$) is now easily recognized from the fact that the label of its parent node no longer contains the index 6, and hence, by step 5.3, the undesired copy of the duplicate of $g(\rho_6)$ (as well as the undesired copy of the duplicate of $g(\rho_8)$) will be deleted.

Acknowledgments. The authors are indebted to Peter Peelman who produced the figures in this article and to Jan Van den Bussche whose critical comments on a previous version of this paper were very helpful.

REFERENCES

[1] S. Abiteboul and N. Bidoit, *Non first normal form relations to represent hierarchically organized data*, in Proceedings 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press, New York, 1984, pp. 191–198.

[2] S. Abiteboul and R. Hull, IFO: *A formal semantic database model*, ACM Trans. Database Systems, 12 (1987), pp. 525–565.

[3] A. Aho, J. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[4] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Schmueli, and S. Tsur, *Sets and negation in a logic database language* (LDL1), in Proceedings 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM Press, New York, 1987, pp. 21–37.

[5] N. Bidoit, *The Verso algebra or how to answer queries with fewer joins*, J. Comput. System Sci., 35 (1987), pp. 321–364.

[6] A. K. Chandra and P. M. Merlin, *Optimal implementations of conjunctive queries in relational databases*, in Proceedings 9th Annual ACM Symposium on the Theory of Computing, ACM Press, New York, 1977, pp. 77–90.

[7] P. Chen, *The Entity-Relationship Model: Toward a unified view of data*, ACM Trans. Database Systems, 1 (1976), pp. 9–36.

[8] E. F. Codd, *Extending the database relational model to capture more meaning*, ACM Trans. Database Systems, 4 (1979), pp. 397–434.

[9] P. Dadam, F. Kuespert, F. Andersen, H. Blanken, R. Erbe, J. Guenauer, V. Lum, P. Pistor, and G. Walch, *A* DBMS *prototype to support extended* NF2 *relations: An integrated view on flat tables and hierarchies*, in Proceedings ACM SIGMOD International Conference on Management of Data, ACM Press, New York, 1986, pp. 356–364.

[10] S. Ginsburg, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.

[11] G. H. Gonnet and F. W. Tompa, *Mind your grammar: A new approach to modelling text*, Technical Report, University of Waterloo, 1988.

[12] M. Hammer and D. McLeod, *Database description with* SDM: *A semantic database model*, ACM Trans. Database Systems, 6 (1981), pp. 351–386.

[13] R. Hull and C. Yap, *The Format Model: A theory of database organization*, J. Assoc. Comput. Mach., 31 (1984), pp. 518–537.

[14] G. Jaeschke and H.-J. Schek, *Remarks on the algebra on non first normal form relations*, in Proceedings 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press, New York, 1982, pp. 124–138.

[15] D. Janssens and G. Rozenberg, *On the structure of node label controlled languages*, Inform. Sci., 20 (1980), pp. 191–216.

[16] G. M. Kuper and M. Y. Vardi, *A new approach to database logic*, in Proceedings 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, ACM Press, New York, 1984, pp. 124–138.

[17] D. Maier, *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[18] J. Paredaens, P. De Bra, M. Gyssens, and D. Van Gucht, *The Structure of the Relational Database Model*, EATCS Monographs on Theoretical Computer Science, 17, Springer-Verlag, Berlin, 1989.

[19] D. Shipman, *The Functional Data Model and the data language* DAPLEX, ACM Trans. Database Systems, 6 (1981), pp. 140–173.

[20] S. J. Thomas and P. C. Fischer, *Nested relational structures*, in Advances in Computing Research III: The Theory of Databases, P. C. Kanellakis, ed., JAI Press, Greenwich, CT, 1986, pp. 269–307.

[21] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Vols. I and II, Computer Science Press, Rockville, MD, 1988, 1989.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1138–1151, December 1994

SELECTING HEAVILY COVERED POINTS*

BERNARD CHAZELLE†, HERBERT EDELSBRUNNER‡, LEONIDAS J. GUIBAS§, JOHN E. HERSHBERGER¶, RAIMUND SEIDEL‖, AND MICHA SHARIR**

Abstract. A collection of geometric *selection lemmas* is proved, such as the following: For any set P of n points in three-dimensional space and any set $\mathcal{S}$ of m spheres, where each sphere passes through a distinct point pair in P, there exists a point x, not necessarily in P, that is enclosed by $\Omega(m^2/(n^2 \log^6 \frac{n^2}{m}))$ of the spheres in $\mathcal{S}$. Similar results apply in arbitrary fixed dimensions, and for geometric bodies other than spheres. The results have applications in reducing the size of geometric structures, such as three-dimensional Delaunay triangulations and Gabriel graphs, by adding extra points to their defining sets.

Key words. discrete geometry, computational geometry, selecting points, covering, intervals, boxes, spheres, Delaunay triangulations, finite-element meshes, Gabriel graphs

AMS subject classifications. 05B99, 51M99, 52A99, 68Q20, 68R05

1. Introduction.

The research that led to the results reported in this paper was originally focused on a problem about Delaunay triangulations for finite point sets in three-dimensional space. For such a set $P = \{p_1, p_2, \ldots, p_n\}$, the *Delaunay triangulation*, $\mathcal{D}(P)$, consists of all tetrahedra whose circumscribed spheres enclose no points of P [7], [10], [17]. Depending on how the points are distributed, the number of edges can vary between linear and quadratic in n. Euler's relation for three-dimensional cell complexes implies that the number of triangles and tetrahedra, and therefore the total combinatorial size of $\mathcal{D}(P)$, is proportional to the number of edges. We considered the question whether for every set of n points P there exists a point set Q so that $\mathcal{D}(P \cup Q)$ is guaranteed to have only a small number of edges. This question is motivated by the use of Delaunay triangulations in the discretization of three-dimensional objects [4], for finite-element analysis and related applications, where the size of the analysis has a strong effect on the efficiency of the analysis [18]. Of course, any set of n points in three dimensions admits a linear-size triangulation [10]; however, the Delaunay triangulation is preferred in these applications, because its tetrahedra are, in a certain sense, the most "round" possible, a property that affects the quality of the finite-element analysis.

A fairly intuitive approach to the problem is to identify a point that lies inside a large number of spheres circumscribing the tetrahedra of the current Delaunay triangulation. Adding this point will remove all corresponding tetrahedra and replace them by at most a linear number of new tetrahedra. Thus, the problem of slimming Delaunay triangulations can be attacked by showing that if there are many circumscribing spheres then there must be a point enclosed by many of them. It turns out that this is indeed true, for certain quantifications of "many," and

*Received by the editors April 16, 1990; accepted for publication (in revised form) October 4, 1993. A preliminary version of this paper has appeared in Proc. 6th ACM Symp. on Computational Geometry (1990), pp. 116–127.

†Department of Computer Science, Princeton University, Princeton, New Jersey 08544. The work of this author has been supported by National Science Foundation grant CCR-87-00917.

‡Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. The work of this author has been supported by National Science Foundation grant CCR-87-14565.

§DEC Systems Research Center, Palo Alto, California 94301, and Computer Science Department, Stanford University, Stanford, California 94305.

¶DEC Systems Research Center, Palo Alto, California 94301.

‖Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. The work of this author was supported by National Science Foundation grant CCR-88-09040.

**School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The work of this author has been supported by Office of Naval Research grant N00014-87-K-0129, by National Science Foundation grants DCR-83-20085 and CCR-89-01484, and by grants from the U.S.-Israeli Binational Science Foundation, the Israeli National Council for Research and Development, and the Fund for Basic Research of the Israeli Academy of Sciences.

that similar results can be obtained in more general settings, involving various other geometric objects, in two, three, and beyond three dimensions. We now summarize the main results and present the outline of this paper.

TABLE 1
Summary of combinatorial results on multiply covered points.

objects	dimension	bound	§
intervals	1	$\Omega(m^2/n^2)$	2.1
rectangular boxes	d	$\Omega\left(m^2 \Big/ \left(n^2 \log^{2d-2} \frac{n^2}{m}\right)\right)$	2.2
diameter spheres	d	$\Omega\left(m^2 \Big/ \left(n^2 \log^{2d-2} \frac{n^2}{m}\right)\right)$	3.1
general spheres	d	$\Omega\left(m^2 \Big/ \left(n^2 \log^{2d} \frac{n^2}{m}\right)\right)$	3.2

Sections 2 and 3 present the main results of the paper. They are combinatorial in nature and show how to select multiply covered points in collections of rectangular boxes (§2) and spheres or more general convex bodies (§3). Table 1 lists these results. In each case, the problem is defined for a set of n points in d dimensions, and for a subset of m of the $\binom{n}{2}$ point pairs, where each of these pairs defines a geometric object of some kind. The bound given in the third column of the table is $\Omega(f(n, m))$ if there is always a point enclosed by at least that many of the m objects. In all cases, the bounds are nontrivial only if the number of objects is significantly larger than the number of points.

Sections 4 and 5 discuss the problem of reducing the combinatorial size of certain geometric structures by adding new points. The combinatorial result for general spheres is used in §4 to show, using a constructive proof, that for any set P of n points in three dimensions there is a set Q of $O(n^{1/2} \log^3 n)$ points so that the Delaunay triangulation of $P \cup Q$ has at most $O(n^{3/2} \log^3 n)$ edges. Section 5 studies the case of *Gabriel graphs*. The Gabriel graph of a set P of n points in $d \geq 1$ dimensions, denoted by $\mathcal{G}(P)$, has an edge between two points p and q in P if and only if the sphere whose diameter is pq encloses no point of P. We show that the size of $\mathcal{G}(P)$ in three dimensions can be $\Omega(n^2)$, and that it can be slimmed down by adding extra points, as in the case of Delaunay triangulations.

The idea of adding points to slim down the size of Delaunay triangulations has already been used in a paper of Chew [6], where he triangulates polygons without small angles, by finding sharp triangles in the constrained Delaunay triangulation of the polygon, and by adding new points at their circumcenters. After the original appearance of this paper [5], an improved and fairly complete solution to the slimming problem has been given by Bern, Eppstein, and Gilbert [3] (see also [2]), who showed that, in any fixed dimension, $O(n)$ points can always be added to any given set of n points, to reduce the size of the Delaunay triangulation of the combined set to linear in n. The technique of [3] is not really comparable to the approach taken here, and it does not supercede our main selection lemmas, which, as we believe, provide useful machinery for tackling other, unrelated geometric problems. Indeed, our selection results have been used in a companion paper [1] to derive an improved bound on the number of halving planes of a point set in three dimensions.

2. Selecting a point within rectangular boxes. The primary combinatorial tool used to prove the results of this paper is what we call the "selection lemma" (Lemma 2.1). This section formulates and proves this lemma and demonstrates its generalization to rectangular boxes in

$d \geq 2$ dimensions. Although we phrase the results in geometric terms, they are combinatorial in nature.

2.1. The selection lemma. To state the selection lemma we make the following definition. For two points $p < q$ on the real line we call $\beta_{pq} = \{x \mid p < x < q\}$ the *interval* of $\{p, q\}$. For any set V, we denote by $\binom{V}{2}$ the set of all unordered pairs $\{p, q\}$, for $p \neq q \in V$. The following lemma can also be found in [1], where generalizations different from the ones in this paper are studied.

LEMMA 2.1. *Let V be a set of n points on the real line and let $E \subseteq \binom{V}{2}$ be a set of m edges. For a point x not necessarily in V, let $E(x)$ denote any subset of the edges in E whose intervals contain x, define $m(E(x)) = |E(x)|$, and let $n(E(x))$ be the number of points incident to (i.e., endpoints of) edges in $E(x)$.*

(i) *There is a point x and a set $E(x)$ with $m(E(x)) \geq m^2/4n^2$.*

(ii) *There is a point y for which there is a set $E(y)$ with*

$$m(E(y))/n(E(y)) \geq m/\left(6n \log \frac{n^2}{m}\right).$$

Both bounds are tight up to multiplicative constants.[1]

Proof. We assume that $m \geq 2n$; otherwise both assertions hold trivially. In order to show (i) choose $k - 1$ points, none of which are in V, cutting the line into k intervals so that each contains no more than $\lceil \frac{n}{k} \rceil < \frac{n}{k} + 1$ points of V (k will be specified later). The number of edges whose intervals contain none of the $k - 1$ points is therefore at most $k\binom{\lceil \frac{n}{k} \rceil}{2} < (n^2 + nk)/2k$. Each of the remaining intervals contains at least one of the $k - 1$ points and there are at least $m - (n^2 + nk)/2k$ such intervals, which is at least $\frac{m}{2}$ if we choose $k = \lceil n^2/(m - n) \rceil$. By the pigeonhole principle one of the chosen points is contained in at least $m/2(k - 1) \geq (m^2 - mn)/2n^2 \geq m^2/4n^2$ intervals (it is only in the last inequality that we needed the assumption $m \geq 2n$).

It is easy to see that this bound is tight, up to the multiplicative constant. For given m and n let V consist of about $n^2/2m$ groups of about $\frac{2m}{n}$ consecutive points each, and let E contain only edges within but not across groups. Any point x can only be covered by the intervals within one group and there are at most about m^2/n^2 such intervals covering a common point.

To prove (ii), build an ordered minimum height binary tree whose nodes are the $k - 1$ chosen points (for the same k chosen in (i)), so that the tree inorder gives the points sorted from left to right. The height of the tree is $h = \lfloor \log(k - 1) \rfloor \leq 2 \log n^2/m$, as is easily verified. For a node y define $E(y)$ as the set of edges in E whose intervals contain y but no ancestors of y. In this way each edge whose interval contains at least one of the $k - 1$ points is counted exactly once. By what we said above we therefore have $\sum_y m(E(y)) \geq \frac{m}{2}$. Because each point can be incident to edges of at most one node per level we also have $\sum_y n(E(y)) < n(1 + h)$. Now suppose that $m(E(y))/n(E(y)) < m/(2n(1 + h))$ for each node y. But then

$$\sum_y m(E(y)) < \frac{m}{2n(1 + h)} \sum_y n(E(y)) < \frac{m}{2},$$

which is a contradiction. This implies that there is a point y with $m(E(y))/n(E(y)) \geq m/(2n(1 + h)) \geq m/\left(6n \log \frac{n^2}{m}\right)$.

The remainder of the proof shows that the lower bound in (ii) is tight, up to the multiplicative constant. The argument consists of two steps. For the first step consider the graph defined by the set of points $W = \{1, 2, \ldots, \ell\}$ and the set of edges $F = \{\{i, j\} \mid j - i \text{ is a power of } 2\}$.

[1] All logarithms in this paper are to the base 2.

Notice that $|F| = \Theta(\ell \log \ell)$. We show that the edges whose intervals contain some arbitrary point y form a forest by arguing that these edges cannot form a cycle. So assume there is a cycle of edges $\{i_0, i_1\}, \{i_1, i_2\}, \ldots, \{i_k, i_0\}$ whose intervals all contain y, and let i_0 be the point closest to y (we may assume that y is not an integer multiple of $\frac{1}{2}$ so i_0 is uniquely defined). By definition we have $|i_j y| < |i_{j+1} y|$ for $j = 0$ and we now argue that this is true in general. Assume it is true up to j. Because $|i_j y| < |i_{j+1} y|$ and the lengths of all intervals are powers of 2, $|i_{j+1} i_{j+2}| \geq 2 |i_j i_{j+1}|$ unless $i_{j+2} = i_j$, which is impossible because this would mean that an edge is reused. Consequently, the distances of the i_j from y strictly increase with increasing index, which contradicts the assumption of a cycle. Since every subgraph of a forest is again a forest and since every forest has more vertices than edges the above argument proves that the lower bound in (ii) is asymptotically tight for $m = \Theta(n \log n)$. Nothing has to be proved if m is even smaller than that.

The second step covers other ratios of m and n as follows. For each point $i \in W$ let V contain a group, G_i, of κ consecutive points, for κ some fixed positive integer. We also define $E = \{\{p, q\} \mid p \in G_i,\ q \in G_j,\ \{i, j\} \in F\}$. Now, $n = |V| = \kappa\ell$ and $m = |E| = \Theta(\kappa^2 \ell \log \ell)$ and therefore $\frac{m}{n} = \Theta(\kappa \log \ell)$. We show below that $m(E(y))/n(E(y)) \leq \kappa$ for every point y and every subset $E(y)$ of the set of edges in E whose intervals contain y. But this is equivalent to showing that (ii) is asymptotically tight because

$$\frac{m}{6n \log \frac{n^2}{m}} = \Theta\left(\frac{\kappa \log \ell}{\log \frac{\ell}{\log \ell}}\right) = \Theta(\kappa).$$

To show $m(E(y))/n(E(y)) \leq \kappa$ let $E(y)$ be a subset of the edges whose intervals contain y and let n_i be the number of points in G_i incident to at least one edge in $E(y)$. Define $F(y)$ as the set of pairs $\{i, j\} \in F$ so that $E(y)$ contains an edge $\{p, q\}$ with $p \in G_i$ and $q \in G_j$. Clearly, $m(E(y)) = |E(y)| \leq \sum_{\{i,j\} \in F(y)} n_i n_j$. By the argument of the previous paragraph, $F(y)$ defines a forest which implies the existence of a leaf i whose contribution to $\sum n_i n_j$ is therefore at most $n_i \kappa$. Since we can reduce a forest to the empty graph by repeatedly removing a leaf with its incident edge, we get $\sum n_i n_j \leq \kappa \sum n_i = \kappa n(E(y))$, thus proving that (ii) is asymptotically tight. □

Remarks. (1) Part (ii) of the selection lemma implies an inequality that is only slightly weaker than (i). To see this note that $m(E(y))/n(E(y))^2 \leq 1$, which implies $n(E(y)) \geq m/\left(6n \log \frac{n^2}{m}\right)$ using (ii). Using (ii) again gives $m(E(y)) \geq m^2/\left(36n^2 \log^2 \frac{n^2}{m}\right)$.

(2) The proofs of the lower bounds in the selection lemma are constructive. Assume the graph (V, E) is given with the points sorted from left to right. Point x can be found in time $O(m)$ by a single scan from left to right that keeps track of how many intervals cover the gap between the current two adjacent points. By a slightly more complicated algorithm we can also find a point y satisfying (ii) in time $O(m)$. The idea is to build explicitly the binary tree described in the proof above (see also [9]). We first build the tree in time $O(k)$ and then assign the endpoints of the edges to the gaps between the $k - 1$ points in time $O(m)$ during a left to right scan. From the gaps of its endpoints we get the leftmost and rightmost of the $k - 1$ points that lie in the interval of the edge and we get the lowest common ancestor of the corresponding two nodes, all in constant time (see [14]). It now remains to traverse all nodes of the tree and to select the best one. If the points in V are not presorted then points x and y can be computed in time $O(m + n \log n)$.

2.2. Rectangular boxes. For two points $p = (\pi_1, \pi_2, \ldots, \pi_d)$ and $q = (\phi_1, \phi_2, \ldots, \phi_d)$ in d dimensions we define

$$\beta_{pq} = \{x = (\xi_1, \xi_2, \ldots, \xi_d) \mid \pi_i < \xi_i < \phi_i \text{ or } \phi_i < \xi_i < \pi_i \text{ for } 1 \leq i \leq d\}$$

and call it the *box* of $\{p, q\}$. We now generalize Lemma 2.1 from intervals to boxes in d dimensions.

THEOREM 2.2. *Let V be a set of n points in $d \geq 1$ dimensions, so that no two coordinates of any two points in V are the same. Let $E \subseteq \binom{V}{2}$ be a set of $m \geq 2n$ edges. For a point x not necessarily in V, let $E(x)$ denote any subset of the edges in E whose boxes contain x, define $m(E(x)) = |E(x)|$, and let $n(E(x))$ be the number of points incident to edges in $E(x)$. Then there exists a constant $c_d > 0$ depending only on d such that the following holds.*

(i) *There is a point x and a set $E(x)$ with $m(E(x)) \geq m^2/\left(c_d n^2 \log^{2d-2} \frac{n^2}{m}\right)$.*

(ii) *There is a point y for which there is a set $E(y)$ with*

$$m(E(y))/n(E(y)) \geq m/\left(c_d n \log^d \frac{n^2}{m}\right).$$

Proof. We prove the theorem for $c_d = 6^{2^d-1}$ using induction over d; the base case, $d = 1$, is settled by the selection lemma. We remark that no effort is made to minimize c_d.

If $d \geq 2$ then project all points orthogonally onto the $(d-1)$-dimensional hyperplane $x_d = 0$. By the inductive assumption there is a point y' in this hyperplane and a subset $E(y')$ of the edges in E whose $(d-1)$-dimensional boxes (the projections of the boxes β) contain y' so that

$$\frac{m(E(y'))}{n(E(y'))} \geq \frac{m}{c_{d-1} n \log^{d-1} \frac{n^2}{m}}.$$

The edges whose $(d-1)$-dimensional boxes contain y' are such that their d-dimensional boxes intersect the line parallel to the dth coordinate axis that goes through y'. On this line we have a one-dimensional problem with $m(E(y'))$ intervals defined by $n(E(y'))$ endpoints. The selection lemma thus implies that there are points x and y with

$$m(E(x)) \geq \frac{m(E(y'))^2}{4n(E(y'))^2} \geq \frac{m^2}{c_d n^2 \log^{2d-2} \frac{n^2}{m}}$$

because $4c_{d-1}^2 \leq c_d$, and

$$\frac{m(E(y))}{n(E(y))} \geq \frac{m(E(y'))}{6n(E(y')) \log \frac{n(E(y'))^2}{m(E(y'))}} \geq \frac{m}{c_d n \log^d \frac{n^2}{m}}$$

because $6c_{d-1} \log(n(E(y'))^2/m(E(y'))) \leq 6c_{d-1} \log((n^2/m) \cdot c_{d-1} \log^{d-1}(n^2/m)) \leq c_d \log^d(n^2/m)$ if $d \geq 2$. □

Remarks. (1) Here is a purely combinatorial formulation of Theorem 2.2: Take a graph with vertex set $\{1, 2, \ldots, n\}$ and a set of m edges, and consider d permutations of the vertex set. Then it is possible to cut each permutation into a left and a right part so that there are "many" edges $\{i, j\}$ with i and j separated in each permutation. How many such edges there are is quantified as in Theorem 2.2.

(2) A noninductive proof of Theorem 2.2 can be given by choosing some k points in d dimensions and then using the pigeonhole principle directly. If the point set is based on the so-called d-fold rectangle or interval tree [9] then the same bounds as above can be derived.

(3) We have seen that the lower bounds of the (one-dimensional) selection lemma are tight up to the multiplicative constants. This is equivalent to saying that Theorem 2.2 is asymptotically tight for $d = 1$. Are the bounds of Theorem 2.2 asymptotically tight also for $d \geq 2$?

(4) Note that (ii) implies (i) up to a polylogarithmic factor. This is because $m(E(y))/(n(E(y))^2) \leq 1$ and therefore $n(E(y)) \geq m/\left(c_d n \log^d \frac{n^2}{m}\right)$ using (ii). Using (ii) again gives $m(E(y)) \geq m^2/\left(c_d^2 n^2 \log^{2d} \frac{n^2}{m}\right)$.

(5) Given a graph (V, E) with the points sorted along each axis, a point y satisfying Theorem 2.2 (ii) can be computed in time $\mathrm{O}(m)$. The algorithm that finds y within this time bound iterates the one-dimensional algorithm mentioned in remark (2) after the selection lemma, once for each dimension. A point x satisfying (i) can be constructed in the same amount of time. If no presorting is assumed then the time to find points x and y is $\mathrm{O}(m + n \log n)$.

3. Selecting a point within spheres. This section extends the selection lemma to circles, spheres, and other geometric objects. In §3.1 we consider spheres defined by antipodal point pairs. In §3.2 we generalize the result to the case where the sphere defined by two points is arbitrary as long as it passes through the two points. We say that a sphere *encloses* a point, or the point lies *inside* the sphere, if the point belongs to the open ball bounded by the sphere. Section 3.3 studies a sufficient but fairly general condition that allows a similar result as for spheres. Finally, §3.4 presents a curious application of our methods to a problem about points and angles.

3.1. Diameter spheres. Let V be a set of n points in $d \geq 2$ dimensions. The *diameter sphere* of a point pair $\{p, q\}$, δ_{pq}, for $p, q \in V$, is the smallest $(d-1)$-sphere that passes through both points. Thus, $z = (p+q)/2$, the midpoint between p and q, is its center and $\rho = \frac{|pq|}{2}$, half the distance between p and q, is its radius. Observe that for all points x in the box β_{pq} the distance to z is smaller than ρ. In other words, β_{pq} is enclosed in δ_{pq}. Moreover, if we rotate the coordinate axes, as necessary, we may assume that no two coordinates of any two distinct points in V are the same. The following result is therefore an immediate corollary of Theorem 2.2.

COROLLARY 3.1. *Let V be a set of n points in $d \geq 2$ dimensions and let $E \subseteq \binom{V}{2}$ denote any set of $m \geq 2n$ edges. For a point x not necessarily in V, let $E(x)$ be a subset of the edges whose diameter spheres enclose x, let $m(E(x)) = |E(x)|$, and let $n(E(x))$ be the number of points incident to edges in $E(x)$.*

(i) *There is a point x and a set $E(x)$ with $m(E(x)) \geq m^2/\left(c_d n^2 \log^{2d-2} \frac{n^2}{m}\right)$.*

(ii) *There is a point y for which there is a set $E(y)$ with*

$$m(E(y))/n(E(y)) \geq m/\left(c_d n \log^d \frac{n^2}{m}\right).$$

Remark. This result can also be interpreted in terms of angles $\angle pxq$, where p and q are points of V and x is an observation point. We consider all pairs $\{p, q\}$ and thus set $m = \binom{n}{2}$. Point x lies inside δ_{pq} if and only if $\angle pxq > \frac{\pi}{2}$. Thus, Corollary 3.1 implies that it is always possible to find a point x so that $\Omega(n^2)$ point pairs define an obtuse angle at x. Section 3.4 will elaborate on this interpretation and show a similar result for angles larger than $\frac{\pi}{2}$.

3.2. General spheres. Next we extend the result for diameter spheres to general spheres. For this extension we let V be a set of n points in $d \geq 2$ dimensions and E be a set of undirected edges between the points as usual. For each edge $\{p, q\} \in E$ we let σ_{pq} be an arbitrary but fixed $(d-1)$-sphere that passes through p and q. Unless $\sigma_{pq} = \delta_{pq}$, σ_{pq} intersects δ_{pq} in a great-$(d-2)$-sphere of δ_{pq}. Therefore, exactly half of δ_{pq} is enclosed by σ_{pq} and at least half of the ball bounded by δ_{pq} lies inside σ_{pq}. If we are lucky then point x (or y) of Corollary 3.1 lies in the halves enclosed by the spheres σ for a constant fraction of the diameter spheres.

In this case, the bounds of Corollary 3.1 are the same, up to a constant multiplicative factor, as for general spheres. Otherwise, almost all spheres do not contain x. We call σ_{pq} *anchored* if this is the case, that is, x does not lie inside σ_{pq} but it lies inside δ_{pq}. All anchored spheres must lie fairly close to x in the sense that the cone with apex x tangent to any such sphere has opening angle at least $\frac{\pi}{2}$. We will show how to select another point that is guaranteed to lie inside many of the anchored spheres. More precisely, we show the following theorem.

THEOREM 3.2. *Let V be a set of n points in $d \geq 2$ dimensions, and let $E \subseteq \binom{V}{2}$ be a set of $m \geq 2n$ edges. For a point x not necessarily in V let $m(E(x))$ be the number of edges whose spheres enclose x.*

(i) *There is a point x with $m(E(x)) \geq m^2/\left(c_d'' n^2 \log^{2d} \frac{n^2}{m}\right)$, where c_d'' is a positive constant that depends only on d.*

(ii) *There is a point y and a subset $E(y)$ of the edges in E whose spheres enclose y so that*

$$\frac{m(E(y))}{n(E(y))} \geq \frac{m}{c_d''' n \log^{d+1} \frac{n^2}{m}},$$

where $m(E(y))$ and $n(E(y))$ are defined as usual and c_d''' is some positive constant.

Proof. We prove only (i); claim (ii) can be proved in a similar manner, using Lemma 2.1 (ii) instead of (i). Let y be a point that lies inside many diameter spheres of the edges in E, where "many" is quantified as in Corollary 3.1 (ii). Thus, there is a subset $E(y)$ of the edges in E whose diameter spheres enclose y so that

$$\text{(1)} \qquad \frac{m(E(y))}{n(E(y))} \geq \frac{m}{c_d n \log^{d} \frac{n^2}{m}},$$

where $m(E(y)) = |E(y)|$ and $n(E(y))$ is the number of points incident to edges in $E(y)$. Let $\mathcal{S}$ be the set of spheres of edges in $E(y)$ that do not enclose y; so all spheres in $\mathcal{S}$ are anchored and we can assume that $|\mathcal{S}| \geq \frac{m(E(y))}{2}$.

To argue about y's view of the world we consider a sphere σ_y with center y and centrally project all centers of spheres in $\mathcal{S}$ onto σ_y. We can assume that no two centers project onto the same point on σ_y. Define a *cap* of σ_y as its intersection with a closed cone with apex y whose opening angle is $\frac{\pi}{6}$, that is, the cone consists of all points p so that the angle between the cone's axis and the half-line through p that starts at y is at most $\frac{\pi}{12}$. By a standard compactness argument, σ_y can be covered by a finite (i.e., constant) number, c_d', of caps [12]. Therefore, there exists a cap that contains a constant fraction of the projected centers. Let R be the half-line that is the axis of the corresponding cone C_R and let $\mathcal{S}_R$ be the set of spheres in $\mathcal{S}$ whose centers lie in C_R (that is, project to points in the cap). Since the opening angle of the cone with apex y tangent to any sphere σ in $\mathcal{S}_R$ is at least $\frac{\pi}{2}$, it easily follows that R intersects σ in two points which delimit an interval that is at least as long as the radius of σ. To see this it suffices to consider the two-dimensional cross section of σ with the plane spanned by R and by the center of σ. In this plane, the angle δ between R and the tangent from y to σ that is nearer to R (see Fig. 1) is at least $\frac{\pi}{4} - \frac{\pi}{12} = \frac{\pi}{6}$. However, $\delta = \frac{1}{2}s - \frac{1}{2}s'$, where s and s' are the two arcs of σ, measured in radians, delimited between R and the tangent line. In particular, this implies that the smaller arc cut off σ by R is $s + s' \geq \frac{\pi}{3}$, from which it follows trivially that R intersects σ in a chord whose length is at least the radius of σ.

At this point we face a one-dimensional problem on R. Intersect R with all open balls bounded by spheres in $\mathcal{S}_R$. This gives a set of at least $m(E(y))/(2c_d')$ intervals, and we want to show, using the selection lemma, that there is a point in many such intervals and therefore inside many spheres. The difficulty we have to cope with is that the intervals can have many

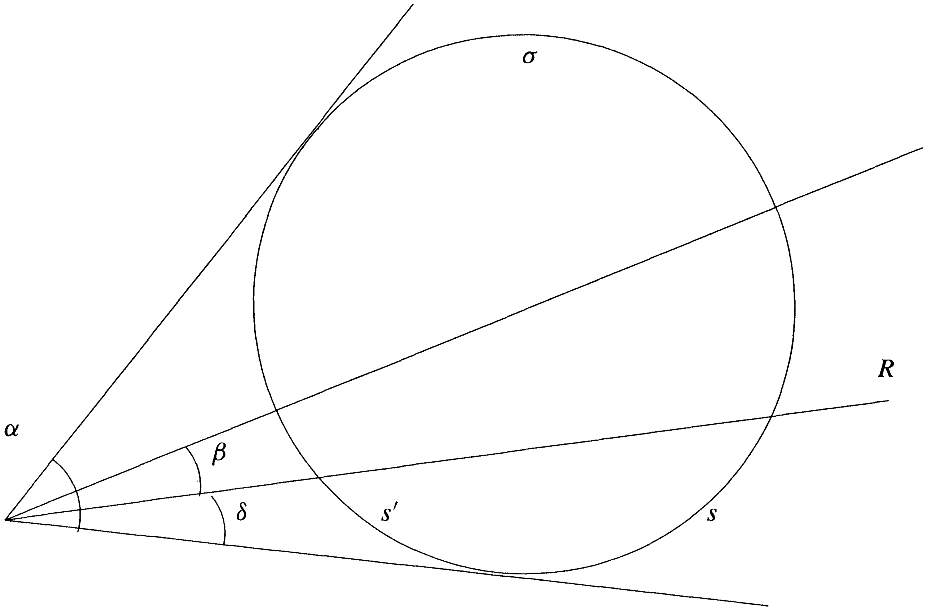


FIG. 1. *R intersects σ in a long chord.*

more than $n(E(y))$ endpoints. In fact, most likely there are twice as many endpoints as there are intervals. We show below that it is possible to replace each interval by an interval contained in it so that the total number of endpoints of the new intervals is at most $6n(E(y))$. Using Lemma 2.1 (i) it follows then that there is a point x contained in

$$m(E(x)) \geq \frac{m(E(y))^2}{4(2c'_d)^2(6n(E(y)))^2}$$

intervals. Together with (1) this implies

$$m(E(x)) \geq \frac{m^2}{c''_d n^2 \log^{2d} \frac{n^2}{m}},$$

where $c''_d = (24c_d c'_d)^2$.

We now show how to reduce the number of endpoints to $6n(E(y))$. Take all spheres in $\mathcal{S}_R$ that go through a common point $p \in V$ and intersect them with the (two-dimensional) plane h that contains R and p. Let $\sigma \in \mathcal{S}_R$ go through p and denote by $\bar{\sigma}$ the closed ball bounded by σ. Clearly, the radius of the circle $h \cap \sigma$ is smaller than or equal to the radius of σ. Furthermore, the interval $R \cap \bar{\sigma}$ is at least as long as the radius of σ because of the way R is chosen. Let a and b be the endpoints of this interval. Then the angle $\angle apb$ is at least $\frac{\pi}{6}$ (see Fig. 2). Hence, 12 half-lines starting at p suffice to stab all these angles, and at most six of them intersect R. These at most six half-lines stab all intervals of the form $R \cap \bar{\sigma}_{pq}$ with $\sigma_{pq} \in \mathcal{S}_R$, p fixed, and q arbitrary.

For the final argument we place at most six points for each one of the $n(E(y))$ points incident to edges in $E(y)$, which gives at most $6n(E(y))$ points on R. The interval $R \cap \bar{\sigma}_{pq}$ is guaranteed to contain at least one of the at most six points generated by p and at least one of the at most six points generated by q. We can thus replace $R \cap \bar{\sigma}_{pq}$ by one of the at most

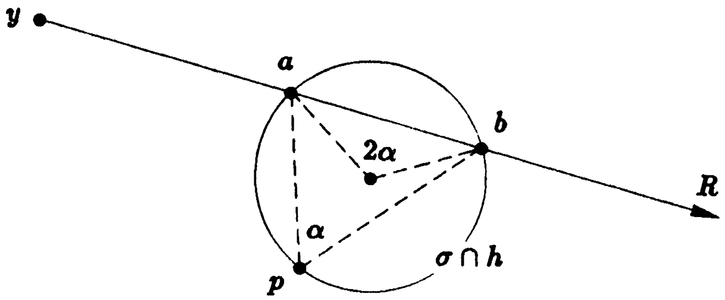


FIG. 2. *The angle formed by a, p, and b is equal to half the angle formed by a, the center of the circle, and b. Since the interval ab is at least as long as the radius of the circle, the latter angle is at least* $\frac{\pi}{3}$.

36 intervals defined by the 12 points generated by p and q and apply the selection lemma as described above. □

Remark. The proof of Theorem 3.2 is constructive and leads to an algorithm that computes a point x with the desired properties in time $O(m + n \log n)$. The first step of this algorithm finds a point y within the required number of diameter spheres (see remark (5) after Theorem 2.2). This takes time $O(m + n \log n)$. Second, a ray R that intersects many anchored spheres sufficiently close to their centers is determined by projecting centers of spheres onto the sphere σ_y around y, covering σ_y with a constant number of caps, and choosing the cap that contains the largest number of projected points. This takes time proportional to the number of projected centers, which is $O(m)$. Finally, the spheres whose centers project onto the chosen cap are intersected with R, thus the defined intervals are replaced by smaller intervals as described, and point x is selected in time $O(m + n \log n)$ in a single scan along R.

3.3. Round objects. A result similar to Theorem 3.2 can be established for a more general class of objects than just spheres. Let p and q be two points in $d \geq 2$ dimensions, let $|pq|$ denote their euclidean distance, and let c_0 and C_0 be two positive constants. A convex set τ_{pq} is said to be (c_0, C_0)-*round* (or simply *round*) for $\{p, q\}$ if

(i) p and q lie on the boundary of τ_{pq}, and

(ii) τ_{pq} contains a d-dimensional ball β_{pq} whose radius is at least $c_0|pq|$ and whose center is at a distance at most $C_0|pq|$ from p and from q.

For example, the ball bounded by the diameter sphere δ_{pq} of p and q is $(\frac{1}{2}, \frac{1}{2})$-round, and it is fairly easy to see that any ball with p and q on its boundary is $(\frac{1}{2}, \frac{\sqrt{2}}{2})$-round. With this definition we can show the following generalization of Theorem 3.2.

THEOREM 3.3. *Let V be a set of n points in $d \geq 2$ dimensions and let $E \subseteq \binom{V}{2}$ be a set of $m \geq 2n$ edges $\{p, q\}$, each associated with an round object τ_{pq}. For a point x not necessarily in V let $m(E(x))$ be the number of edges $\{p, q\}$ with $x \in \tau_{pq}$. Then there is a point x with $m(E(x)) \geq m^2 / \left(cn^2 \log^{2d} \frac{n^2}{m}\right)$, where c is a positive constant that depends on d, c_0, and C_0.*

Proof. To describe where this proof differs from the one of Theorem 3.2 we introduce two auxiliary objects: the ball β'_{pq} and the cone γ_{pq}. The ball β'_{pq} has the same center as β_{pq} and its radius is half of the radius of β_{pq}; the cone γ_{pq} is the convex hull of β_{pq} and p (see Fig. 3). Clearly, we have $\tau_{pq} \supseteq \gamma_{pq} \supseteq \beta_{pq} \supseteq \beta'_{pq}$.

When we construct the half-line R out of point y (defined as in the proof of Theorem 3.2), we make sure it intersects many of the balls β'_{pq} associated with edges in $E(y)$. Because of condition (ii), R can be found so that it intersects at least a constant fraction of the β'_{pq}.

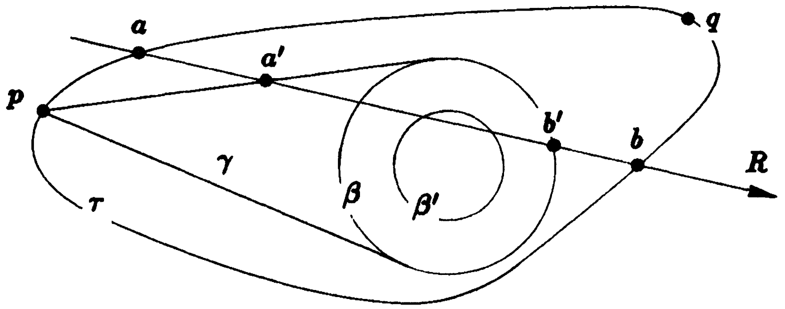


FIG. 3. *The edge $\{p, q\}$ defines an round object τ that contains γ, β, and β'. The half-line R intersects β'; its intersection with γ is $a'b'$ and with τ it is ab.*

Let us now fix our attention on a particular $\tau = \tau_{pq}$ and let a and b be the endpoints of the interval $R \cap \tau$. In order to complete the proof in the same way as the proof of Theorem 3.2 we need to show that the angle $\angle apb$ (and analogously $\angle aqb$) is at least some constant fraction of π.

Notice that the boundary of $\gamma = \gamma_{pq}$ consists of a fan of line segments that form the tangents from p to $\beta = \beta_{pq}$, as well as part of the boundary of β itself (see Fig. 3). Let a' and b' be the endpoints of $R \cap \gamma$; we will prove the stronger result that the angle $\angle a'pb'$ is at least some fixed fraction of π. If one of the points a' or b' lies on one of the line segments that form the tangents from p to β then the result is immediate: the angle subtended at p goes from the boundary of β at least as far as to some point of β'. By condition (ii) the balls β and β' look big from p, so this angle cannot be too small. On the other hand, if both a' and b' lie on the boundary of β then the result follows because $a'b'$ cannot be too short—in particular, it is longer than the radius of β.

We omit all further details, as they are the same as in the proof of Theorem 3.2. □

Remarks. (1) As follows from the above proof, it is not necessary to require that τ_{pq} be convex and that p and q lie on its boundary. All that is needed is condition (ii) and that τ_{pq} contains the cones γ_{pq} and γ_{qp} defined by β_{pq} and points p and q.

(2) It is also interesting to observe that condition (ii) is not sufficient to prove Theorem 3.3. Indeed a counterexample exists already in one dimension. Let $V = \{p_i = 2^i \mid 1 \leq i \leq n\}$ be the set of n points and for $i < j$ define $\tau_{ij} = \{x \mid (2p_i + p_j)/3 < x < (p_i + 2p_j)/3\}$. Thus, τ_{ij} has the same midpoint as the interval β_{ij} delimited by p_i and p_j and its length is one third of that of β_{ij}. However, for any $i < j < k$ we have $\tau_{ij} \cap \tau_{ik} = \emptyset$ because $(2^i + 2^{j+1})/3 < (2^{i+1} + 2^k)/3$. Thus, the set of $\binom{n}{2}$ intervals τ_{ij} can be partitioned into n subsets so that two intervals are disjoint if they belong to the same subset. It follows that there is no point x contained in more than n intervals τ_{ij}.

3.4. A problem about points and angles. For points p and q in d-dimensional space and for angle α, $\frac{\pi}{2} \leq \alpha \leq \pi$, define the *$\alpha$-football* of $\{p, q\}$ as the set

$$\phi_{pq}(\alpha) = \{x \mid \angle pxq \geq \alpha\}.$$

For example, $\phi_{pq}(\frac{\pi}{2})$ is the closed ball bounded by the diameter sphere δ_{pq}, and $\phi_{pq}(\pi)$ is the line segment pq. For general α, $\phi_{pq}(\alpha)$ is the intersection of all closed balls that contain p and q and have a fixed radius depending on $|pq|$ and α. If $\alpha < \pi$ is fixed, then $\phi_{pq}(\alpha)$ contains a ball centered at the midpoint between p and q whose radius is some fixed positive fraction of $|pq|$. Hence, $\phi_{pq}(\alpha)$ is round for $C_0 = \frac{1}{2}$ and $c_0 > 0$ (c_0 goes to zero if α approaches π) and Theorem 3.3 applies. We reformulate this result for the case where *every* pair of points defines an α-football and phrases it in terms of angles.

COROLLARY 3.4. *Let P be a set of n points in $d \geq 2$ dimensions and let $\alpha \geq \frac{\pi}{2}$ be a fixed angle strictly smaller than π. Then there exists a constant c depending on d and α and a point x so that $\angle pxq \geq \alpha$ for at least cn^2 pairs $\{p, q\} \in \binom{P}{2}$.*

Loosely speaking, point x is almost collinear with a constant fraction of the point pairs if α is insignificantly smaller than π, for example $\alpha = 179°$. In other words, x almost lies on each one of a constant fraction of the lines defined by the points.

4. Slimming down spatial Delaunay triangulations. This section deals with Delaunay triangulations for point sets in (three-dimensional) space. Let P be a set of n points in space and let $\mathcal{D}(P)$ be its Delaunay triangulation. For simplicity we assume that no five points are cospherical so that $\mathcal{D}(P)$ is uniquely defined. If this is not the case then it is always possible to enforce it by simulating an arbitrarily small perturbation of the points; see [11]. As mentioned in the introduction, $abcd$ is a tetrahedron of $\mathcal{D}(P)$ if and only if the sphere through points a, b, c, and d does not enclose any points of P.

For $0 \leq i \leq 3$, let f_i be the number of i-dimensional faces of $\mathcal{D}(P)$, that is, $f_0 = n$ is the number of vertices, f_1 is the number of edges, f_2 is the number of triangles, and f_3 is the number of tetrahedra of $\mathcal{D}(P)$. By Euler's relation we have $f_0 - f_1 + f_2 - f_3 = 1$ (see Hopf [15] for an elementary proof of this relation). Because every tetrahedron is bounded by four triangles and every triangle bounds at most two tetrahedra we also have $2f_3 \leq f_2$. This implies

$$f_3 \leq f_1 - n + 1 \quad \text{and} \quad f_2 \leq 2f_1 - 2n + 2. \tag{2}$$

We thus see that f_1, the number of edges of $\mathcal{D}(P)$, is a good measure of the combinatorial complexity of $\mathcal{D}(P)$. We call f_1 the *size* of $\mathcal{D}(P)$.

Depending on how the points are distributed, the size of $\mathcal{D}(P)$ can vary between linear in n and quadratic in n. An extreme example is when the points of P lie on the positive branch of the moment curve, $\mathcal{M} = \{(x, x^2, x^3) \mid x > 0\}$. Because a sphere intersects $\mathcal{M}$ in at most four points, which can be shown using Descartes' sign rule for the polynomial that arises, every point pair defines an edge of $\mathcal{D}(P)$. It follows that the size of $\mathcal{D}(P)$ is $\binom{n}{2}$ (see also [8]). The goal of this section is to show that no matter how badly P is distributed, there is always a small set Q of points in space so that $\mathcal{D}(P \cup Q)$ has size at most $O(n^{3/2} \log^3 n)$.

A sphere is called a *Delaunay sphere* of P if it is the circumscribed sphere of a tetrahedron $abcd$ of $\mathcal{D}(P)$. Using Theorem 3.2 we can show that if there are many Delaunay spheres, then there are many that enclose a common point.

LEMMA 4.1. *Let P be a set of n points in space defining t Delaunay spheres. There is a point x enclosed by $m(E(x)) \geq t^2 / \left(cn^2 \log^6 \frac{n^2}{t}\right)$ Delaunay spheres, for some positive constant c.*

Proof. Note that an edge ab is incident to as many Delaunay spheres as there are tetrahedra in $\mathcal{D}(P)$ that share ab; this number can be as large as $n - 2$. In order to apply Theorem 3.2 we match the edges of $\mathcal{D}(P)$ with the Delaunay spheres so that the Delaunay sphere matched with an edge passes through its endpoints and at least $\frac{t}{6}$ Delaunay spheres have a matching edge. We do this as follows. By definition, each Delaunay sphere is incident to six edges, and, by (2), there are at least t edges. Match an edge with an incident sphere arbitrarily and remove both from further consideration. Thus, there are at most five more edges that can no longer find a matching sphere. If we iterate this process we get at least $\frac{t}{6}$ matched pairs as required.

We thus arrive at a situation where we have n points and $m \geq \frac{t}{6}$ edges with a unique corresponding sphere each. Theorem 3.2 implies that there is a point x enclosed by at least

$$\frac{m^2}{c_3''n^2 \log^6 \frac{n^2}{m}} \geq \frac{t^2}{6^2 c_3'' n^2 \log^6 \frac{6n^2}{t}} \geq \frac{t^2}{cn^2 \log^6 \frac{n^2}{t}}$$

spheres, e.g., for $c = 6^8 c_3''$. □

If we add x to P then all tetrahedra whose circumscribed spheres enclose x disappear by definition. Lemma 4.1 thus implies that it is possible to destroy $\Omega\left(t^2/\left(n^2 \log \frac{n^2}{t}\right)\right)$ tetrahedra at once. However, x also gives rise to new tetrahedra. Because all new tetrahedra are incident to x we can bound their number from above as follows.

LEMMA 4.2. *Let P be a set of n points in space and x a point not in P. Then x is incident to at most $2n - 4$ tetrahedra in $\mathcal{D}(P \cup \{x\})$.*

Proof. Let σ be a sufficiently small sphere with center at x. If we intersect σ with the edges, triangles, and tetrahedra of $\mathcal{D}(P)$ we get a planar graph. Each vertex of this graph corresponds to an edge of $\mathcal{D}(P)$, and if σ is sufficiently small all such edges are incident to x. Because x is incident to at most n edges (at most one per point in P), the planar graph has at most n vertices and, by Euler's relation, at most $2n - 4$ regions. These regions correspond to the tetrahedra incident to x. □

What we said about Delaunay triangulations in space suggests the following algorithm for reducing the size of a Delaunay triangulation by adding points at well-chosen locations. Recall that $m(E(x))$ is the number of Delaunay spheres destroyed by adding point x.

Input. A set P of n points in space.
Output. A set Q of points in space so that $\mathcal{D}(P \cup Q)$ has at most $O(n^{3/2} \log^3 n)$ edges.
Algorithm.

```
Construct $\mathcal{D}(P)$ and set $Q := \emptyset$;
loop find a point $x$ that maximizes $m(E(x))$ in $\mathcal{D}(P \cup Q)$;
    if $m(E(x)) \geq 4n$ then $Q := Q \cup \{x\}$ and update $\mathcal{D}(P \cup Q)$ accordingly
                         else exit
    endif
forever.
```

Using Lemmas 4.1 and 4.2, one can establish the following result.

THEOREM 4.3. *For any set of n points P in three-dimensional space there is a set Q of at most $O(n^{1/2} \log^3 n)$ points so that the Delaunay triangulation of $P \cup Q$ has at most $O(n^{3/2} \log^3 n)$ edges. Such a set Q can be computed in time $O(n^2 \log^7 n)$.*

We omit here details of the analysis, because this result is less significant now, in view of the recent results of Bern et al. [3]. Interested readers are referred to an earlier and fuller version of this paper [5].

5. The size of Gabriel graphs. The Gabriel graph of a finite point set is a subgraph of the Delaunay triangulation that has applications in zoology and geography [13], [16]. Let P be a set of n points in $d \geq 1$ dimensions. The Gabriel graph of P, denoted by $\mathcal{G}(P)$, has an edge between two points p and q in P if and only if their diameter sphere, δ_{pq}, encloses no point of P. The definition implies that the edges of $\mathcal{G}(P)$ are a subset of the edges of the Delaunay triangulation. Thus, $\mathcal{G}(P)$ has only $O(n)$ edges when $d \leq 2$, and trivially at most $O(n^2)$ edges, otherwise. The bound is tight for $d \leq 2$, since each point is incident to at least one edge. The following lemma shows that the bound is also tight for $d > 2$.

LEMMA 5.1. *The maximum number of edges of the Gabriel graph of n points in $d \geq 3$ dimensions is $\Omega(n^2)$.*

Proof. We exhibit a set P of $2n$ points in three dimensions such that $\mathcal{G}(P)$ has at least n^2 edges. Embedding this example in higher-dimensional space proves the lemma for $d > 3$.

We place the points in two groups $\{a_i\}$ and $\{b_j\}$ on interlocking, orthogonal circles. Each circle passes through the center of the other, and the points on each circle are located near the center of the other circle. Each circle has radius 2. The points a_i lie near $(0, 1, 0)$ on a circle in the xy-plane centered on $(0, -1, 0)$. The b_j lie near $(0, -1, 0)$ on a circle in the yz-plane centered on $(0, 1, 0)$. To quantify "nearness" we use a small parameter ϵ:

$$a_i = (f(i), 1 - i\epsilon, 0) \quad \text{and} \quad b_j = (0, -1 + j\epsilon, f(j)),$$

where $1 \leq i, j \leq n$ and $f(k) = \sqrt{4k\epsilon - k^2\epsilon^2} < \sqrt{4k\epsilon}$. We show that for $\epsilon > 0$ sufficiently small, the diameter sphere determined by a pair $\{a_i, b_j\}$ contains no other points of P. The center of the sphere is

$$c_{ij} = \frac{a_i + b_j}{2} = \frac{1}{2}(f(i), \epsilon(j - i), f(j)).$$

We prove that the distance from c_{ij} to a point a_k (or b_k) is minimized when $k = i$ $(k = j)$. The square of the distance is

$$\begin{aligned}(a_k - c_{ij})^2 &= \frac{1}{4}((2f(k) - f(i))^2 + (2 - 2k\epsilon - j\epsilon + i\epsilon)^2 + f(j)^2)\\ &= \frac{1}{4}(16k\epsilon - 4f(k)f(i) + 4i\epsilon + 4 - 8k\epsilon - 4j\epsilon + 4i\epsilon + 4j\epsilon) + \mathrm{O}(\epsilon^2)\\ &= 1 + 2\epsilon(k + i) - f(k)f(i) + \mathrm{O}(\epsilon^2).\end{aligned}$$

Because $f(k)f(i) = 4\epsilon\sqrt{ki} + \mathrm{O}(\epsilon^2)$, we have

$$(a_k - c_{ij})^2 = 1 + 2\epsilon(k - 2\sqrt{ki} + i) + \mathrm{O}(\epsilon^2) = 1 + 2\epsilon(\sqrt{k} - \sqrt{i})^2 + \mathrm{O}(\epsilon^2).$$

For ϵ small enough, this quantity is minimized only when $k = i$. □

We can use Corollary 3.1 to reduce the size of Gabriel graphs. In three dimensions this gives a better bound than the one for Delaunay triangulations, which is based on Theorem 3.2.

THEOREM 5.2. *For any set of n points P in $d \geq 3$ dimensions there is a set Q of $O(n^{1/2} \log^{d-1} n)$ points so that the Gabriel graph of $P \cup Q$ has at most $O(n^{3/2} \log^{d-1} n)$ edges.*

Proof. Here is a sketch of the proof. By Corollary 3.1, if $m \geq 2n$, m the number of edges of $\mathcal{G}(P \cup Q)$, then there is a point x whose addition to Q deletes $m(E(x))$ edges from $\mathcal{G}(P \cup Q)$, where

$$m(E(x)) \geq \frac{m^2}{c_d n^2 \log^{2d-2} \frac{n^2}{m}}.$$

Adding a point to Q adds at most $|P \cup Q|$ edges to $\mathcal{G}(P \cup Q)$. Using an argument similar to that of §4, one can show that the number of edges of $\mathcal{G}(P \cup Q)$ can be reduced to $\mathrm{O}(n^{3/2} \log^{d-1} n)$ by adding points to Q. By reasoning similar to that used in the proof of Theorem 4.3, one can show that the algorithm of §4, modified for Gabriel graphs, produces a set Q of size $\mathrm{O}(n^{1/2} \log^{d-1} n)$. □

REFERENCES

[1] B. Aronov, B. Chazelle, H. Edelsbrunner, L. J. Guibas, M. Sharir, and R. Wenger, *Points and triangles in the plane and halving planes in space*, Discrete Comput. Geom., 6 (1991), pp. 435–442.

[2] M. Bern and D. Eppstein, *Mesh generation and optimal triangulation*, in Computing in Euclidean Geometry, D.-Z. Du and F. K. Hwang, eds., World Scientific, Singapore, 1992, pp. 23–90.

[3] M. Bern, D. Eppstein, and J. Gilbert, *Provably good mesh generation*, in Proc. 31st IEEE Symp. on Foundations of Computer Science, 1990, pp. 231–241; J. Comp. Systems Science, to appear.

[4] J. C. Cavendish, D. A. Field, and W. H. Frey, *An approach to automatic three-dimensional finite element mesh generation*, Internat. J. Numer. Methods Engrg., 21 (1985), pp. 329–347.

[5] B. Chazelle, H. Edelsbrunner, L. J. Guibas, J. Hershberger, R. Seidel, and M. Sharir, *Slimming down by adding; selecting heavily covered points*, Tech Report UIUCDCS-R-90-1574, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, March 1990.

[6] L. P. Chew, *Guaranteed-quality triangular meshes*, Tech. Report TR-89-983, Department of Computer Science, Cornell University, Ithaca, NY, 1989.

[7] B. Delaunay, *Sur la sphère vide*, Izvestia Akademii Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk, 7 (1934), pp. 793–800.

[8] A. K. Dewdney and J. K. Vranch, *A convex partition of R^3 with applications to Crum's problem and Knuth's post-office problem*, Utilitas Math., 12 (1977), pp. 192–199.

[9] H. Edelsbrunner, *A new approach to rectangle intersections, part* I, Intern. J. Comput. Math., 13 (1983), pp. 209–219.

[10] ———, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, Germany, 1987.

[11] H. Edelsbrunner and E. P. Mücke, *Simulation of Simplicity: a technique to cope with degenerate cases in geometric algorithms*, ACM Trans. Graphics, 9 (1990), pp. 66–104.

[12] L. Fejes Tóth, *Lagerungen in der Ebene, auf der Kugel und im Raum*, second ed., Springer-Verlag, Berlin, 1972.

[13] K. R. Gabriel and R. R. Sokal, *A new statistical approach to geographic variation analysis*, Systematic Zoology, 18 (1969), pp. 259–278.

[14] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[15] H. Hopf, *Über Zusammenhänge zwischen Topologie und Metrik im Rahmen der elementaren Geometrie*, Mathematisch-Physikalische Semester Berichte, 3 (1953), pp. 16–29.

[16] D. W. Matula and R. R. Sokal, *Properties of Gabriel graphs relevant to geographic variation research and clustering of points in the plane*, Geographical Analysis, 12 (1980), pp. 205–222.

[17] F. P. Preparata and M. I. Shamos, *Computational Geometry—an Introduction*, Springer-Verlag, New York, 1985.

[18] G. Strang and G. J. Fix, *An Analysis of the Finite Element Method*, Prentice Hall, Englewood Cliffs, NJ, 1973.

SIAM J. COMPUT.
Vol.23, No.6, pp. 1152–1178, December 1994

DISTRIBUTED ALGORITHMS FOR UNIDIRECTIONAL NETWORKS*

YEHUDA AFEK[†] AND ELI GAFNI[‡]

Abstract. This paper addresses the question of distributively computing over a strongly connected unidirectional data communication network. In unidirectional networks the existence of a communication link from one node to another does not imply the existence of a link in the opposite direction. The strong connectivity means that from every node there is a directed path to any other node. The authors assume an arbitrary topology network in which the strong connectivity is the only restriction. Four models are considered, synchronous and asynchronous, and for each node space availability, which grows as either $O(1)$ bits or $O(\log n)$ bits per incident link, where n is the total number of nodes in the network, is considered.

First algorithms for two basic problems in distributed computing in data communication networks, *traversal*, and *election*, are provided. Each of these basic protocols produces two directed spanning trees rooted at a distinguished node in the network, one called *in-tree*, leading to the root, and the other, *out-tree*, leading from the root. Given these trees, the authors efficiently transform bidirectional algorithms to run on unidirectional networks, and in particular solve other problems such as the broadcast and echo [E. J. CHANG, *Decentralized Algorithms in Distributed Systems*, Ph.D. thesis, University of Toronto, October 1979] in a way that is more efficient ($O(n^2)$ messages) than direct transformation (which yields $O(nm)$ messages algorithm). The communication cost of the traversal and election algorithms is $O(nm + n^2 \log n)$ bits ($O(nm)$ messages and time), where m is the total number of links in the network. The traversal algorithms for unidirectional networks of finite automata achieve the same cost ($O(nm + n^2 \log n)$ bits) in the asynchronous case, while in the synchronous case the communication cost of the algorithm is $O(mn)$ bits.

Key words. unidirectional, distributed algorithms, traversal, election

AMS subject classification. 68Q22

1. Introduction.

Distributed network protocols research has focused most of its efforts on bidirectional networks. However, unidirectional networks appear more often than expected. Unidirectional networks appear as a result of a failure of a link or a subset that disrupts communication in one direction but not in the other. For example, the modem circuit at one end could fail to receive (or send) data but still be operating correctly in the other direction. Networks with unidirectional links are also found in radio networks with asymmetric transmission matrices due to differences in transmission power of the stations, in fiber optics, and in very-large-scale integration (VLSI) [ELW90].

In this paper we consider both asynchronous and synchronous unidirectional networks. For election, a problem that we solve over these networks, each node needs at least $O(\log n)$ bits of memory, where n is the total number of nodes in the network. Our algorithms require as much. However, in solving the traversal problem we consider two possibilities of memory space availability. In one case a node has memory that grows logarithmically in n. In the other case the memory stays finite. More precisely, in the first model $O(\log\ n + d_v)$ bits of memory are available for the algorithm in node v, where d_v is the degree of node v. In the second model only $O(d_v)$ bits of memory are available at each node. Note that any distributed algorithm for a strongly connected unidirectional network is obviously also an algorithm for a bidirectional network.

1.1. Unidirectional network traversal—background.

In the traversal problem, one node, called the *root*, creates a token, which has to visit all the nodes in the network one at a time.

*Received by the editors May 29, 1992; accepted for publication (in revised form) June 21, 1993.

[†] Computer Science Department, Tel-Aviv University, Ramat-Aviv 69978 Israel (afek@math.tau.ac.il).

[‡] Computer Science Department, University of California, Los Angeles, California 90024. The work of this author was supported by the National Science Foundation Presidential Young Investigator Award under grant DCR84-51396 and matching funds from Xerox Corp. under grant W881111.

When the token arrives at a node the algorithm has to make two decisions: first, whether there are any more unvisited nodes in the network and second, in case there are, to which outgoing neighbor the token should be passed. This is similar to an on-line version of the "Chinese postman" problem: A postman is placed in a new city in which some or all the streets are one way. The postman has to traverse all the streets of the city but he has no map or any global information on the topology of the city. The question is to devise a strategy by which he will traverse the minimum number of streets, retracing streets as few times as possible.

There are two ways in which we can view the token. One view is that of a mail letter. It is going from place to place. The decisions about the route are made by a sequence of local post offices, while the letter may carry on it some "control information," e.g., an address. This view supports the traversal as handing over a token from node to node. The other is that of the postman above. It is the actual route decision-maker who passes from node to node, while being allowed to leave "marks" in various nodes. This view supports viewing traversal as a process (single thread of control program) that passes from node to node reading from and writing to the nodes. For convenience we will use these two views interchangeably.

Clearly, to be able to traverse a bidirectional network, the network has to be connected. Similarly, to traverse a unidirectional network, the network has to be strongly connected, i.e., there should be a directed path from every node to every other node. Message complexity of $\Omega(m)$ is obviously a lower bound to any traversal algorithm, where m is the total number of links in the network, since every link in the network has to be traversed. Otherwise an untraversed subnetwork could reside in the midst of any untraversed link.

The problem of distributively traversing a bidirectional network has been thoroughly investigated. One such obvious algorithm is based on the depth-first-search (DFS) algorithm [Tar72], [HT73]. In the resulting DFS traversal the process makes $2 \cdot m$ hops, and the number of memory bits it uses at each node is linear in the degree of the node.

In solving the unidirectional traversal problem we would like to adapt the bidirectional DFS traversal. However, DFS employs *backtracking*, i.e., a step in which the DFS process backtracks a link previously traversed in the forward direction. It is not obvious how to emulate backtracking in a unidirectional network. One of the contributions of this paper is such a backtracking scheme. It is facilitated by constructing, on the fly, a structure called an *in-directed forest*. With the help of the in-directed forest the process identifies a path that brings it from the head of the link which is to be backtracked to its tail. The construction of the in-directed forest is reminiscent of the strongly connected components algorithm of Hopcroft and Tarjan [HT73] (see also [Eve79]).

In §2, three traversal algorithms are presented. Traversal-1 is simple but inefficient. In many network instances the process of Traversal-1 makes an exponential number of link traversals before terminating. Traversal-2, which is based on the DFS algorithm, makes at most $O(n \cdot m)$ link traversals on any network (i.e., its time complexity is $O(n \cdot m)$). Furthermore, we show that $\Omega(n \cdot m)$ is, in general, a lower bound on the number of link traversals (time complexity). Both in Traversal-1 and Traversal-2 $O(\log n + d_v)$ bits of memory are required at each node v, and the traversing process carries along $O(\log n)$ bits. Both achieve the same complexity for synchronous and asynchronous networks.

In some applications, such as VLSI, memory size and message length are restricted and the question then arises, could a unidirectional traversal be implemented using only a constant number of bits in every node and with the traversing process carrying some finite amount of information (i.e., in a unidirectional network of finite automata).

Traversal-3 is the variant of Traversal-2 for finite automata network. In the asynchronous case it makes at most $O(n \cdot m + n^2 \cdot \log n)$ link traversals, which is optimal in the worst case (dense networks, in which $m = \Omega(n \log n)$). In the synchronous case it makes at most $O(n \cdot m)$ link traversals. Traversal-3 in the case of a unidirectional ring is an example of a problem whose complexity depends on whether the underlying system is synchronous or asynchronous [AFL83].

All the traversal algorithms yield a spanning infrastructure of the network consisting of two rooted spanning trees, one called an out-tree, in which paths are directed away from the root, and another called an in-tree, in which paths are directed to the root.

1.2. Unidirectional network election—background. In §3 we present a distributed algorithm for election in strongly connected unidirectional networks. The algorithm requires $O(\log n)$ bits of memory in each processor and its communication complexity is $O(n \cdot m + n^2 \log n)$ bits.

To design an election algorithm for strongly connected networks we employ Traversal-2. As a first attempt, consider the following straightforward approach: every initiator starts a traversal. Whenever a lower ID traversal learns about the existence of a higher ID one, it stops. The worst case communication complexity of this algorithm is $O((n \cdot m + n^2 \cdot \log n) \cdot n)$ bits, since $O(n)$ traversals could be initiated such that each spends $O(n \cdot m + n^2 \cdot \log n)$ bits. Clearly this approach is inefficient. As a second attempt, one can improve on it by using the modular technique of Korach, Kutten, and Moran [KKM85] to economically eliminate traversals. Using their technique, the communication complexity is reduced to $O((n \cdot m + n^2 \cdot \log n) \cdot \log n)$ bits, which is still a logarithmic factor away from our complexity. The election algorithm we design employs Traversal-2 as the underlying mechanism but uses it in a more sophisticated way to achieve a communication complexity of $O(nm + n^2 \log n)$ bits and $O(nm)$ time.

1.3. Related work. Although most of the results in this paper were derived in [GA84], before other works explicitly investigated unidirectional networks [ELW90], [GK84], [Kut88], [Kut84] (with the exception of [Kob78]), two bidirectional algorithms, the shortest path algorithm of Gallager [Gal76] and the connectivity checking algorithm of Segall [Seg83], both easily suggest unidirectional election algorithms. In [Seg83], Segall presents a connectivity checking algorithm in which at termination every node knows the IDs of all the other nodes in its connected component. The shortest path algorithm in [Gal76] exhibits the same property when it terminates. The communication complexity of the two algorithms is $O(n \cdot m \cdot \log n)$ bits, and each node is assumed to have $O(n \log n)$ bits of memory.

The suggested unidirectional variation of the two algorithms proceeds in two phases. In the first phase, every node acquires the IDs of its incoming neighbors; in the second, it acquires the IDs of all the other nodes in the network.

More specifically, let an *incoming* neighbor of node v be a node at the other end of an incoming link of v, and let *in-neighbors* of v be the set of all the incoming neighbors of v. Let the *record* of node v be a two-field data structure, of which the first contains the ID of v and the second the IDs of v's in-neighbors. In the first phase, every node transmits its ID on all its incident outgoing links and receives the IDs of its in-neighbors. In the second phase, every node broadcasts its record, via flooding, to all the other nodes in the network. For this purpose, each node v maintains two sets of IDs, the *received* set and the *known* set. The received set contains the IDs of the nodes whose records were already received by v. The known set contains IDs which appeared in a record of at least one node from the received set, i.e., IDs of nodes whose existence is known to v. Initially, at the beginning of the second phase, the received set of node v contains the ID of v, and the known set contains the IDs of v and of v's in-neighbors. Clearly, when the two sets in a node are identical, they contain the IDs of all the nodes in the network (which can easily be proved by induction).

The communication complexity of the algorithm thus described is $O(m^2 \cdot \log n)$ bits; however, assuming that messages sent over one link are received in the order transmitted, i.e., FIFO links, the communication complexity can be reduced to $O(n \cdot m \cdot \log n)$ bits by avoiding repeated transmission of the same ID over the same link. Note that for these algorithms each node is assumed to have $O(n \cdot \log n)$ bits of memory. Similar adaptation and analysis applies to [Gal76].

Our election algorithm is thus an improvement on the algorithms of Gallager and Segall in terms of both communication complexity and the number of memory bits required at each node. Furthermore, our algorithm produces the infrastructure of the in-tree and out-tree mentioned above, which neither Segall's FIFO link case nor Gallager's algorithms provide.

Following publication of this work Kutten [Kut84] independently developed a traversal algorithm for unidirectional networks. He considered only the logarithmic memory asynchronous model, for which he derived an $O(nm)$ messages algorithm [Kut88]. Gafni and Korfhage [GK84] designed an election algorithm for unidirectional Eulerian networks whose message complexity is $O(m \cdot \log n)$. A preliminary version of most of the results reported in this paper appear in [GA84].

More recently Even, Litman, and Winkler [ELW90] proposed a new approach to the problem of traversing a unidirectional synchronous network of finite automata. Assuming that the degree of each node in the network is constant (implying $M = O(n)$) they derive a traversal algorithm that runs in $O(n^2)$ time. They also study the firing squad problem in the synchronous network of finite automata, which is not addressed in this paper.

Interestingly, the results in this paper and in [ELW90] suggest that the unidirectional network of finite automata is inherently sequential. In the traversal algorithm of [ELW90] more than one message is sent in the network in a given time. However, it does not improve the time complexity or the communication complexity of our synchronous traversal algorithm for a network of finite automata, which sends at most one message in the network at any given time. Furthermore, notice that for increased space complexity at each node we were not able to improve the time complexity of the traversal algorithm. However, for the election problem and when the space at the nodes is increased to $O(n \log n)$ bits, the methods of Gallager and Segall can be used to improve the time complexity of our election algorithm but have worse communication complexity. It is interesting to consider whether there is a spectrum of time-space trade-off here.

Perhaps the first paper to consider the traversal problem in a unidirectional network of finite-automata was that of Kobayashi [Kob78], in which he presented an exponential complexity traversal algorithm that is based on the directed depth-first-search idea.

Organization of the paper. In the next section we present four traversal algorithms and, in §3, an election algorithm. In §4 we show how the traversal algorithms may be derived from the election algorithm and we discuss other applications of the various algorithms. Concluding remarks are given in §5.

2. Traversal of unidirectional networks.

2.1. Traversal-1: A simple traversal algorithm.

The first traversal algorithm presented is similar to the centralized algorithm of Fraenkel [Fra70] and is composed of two mechanisms: a termination detection mechanism, and a routing mechanism. The termination detection mechanism enables the traversing process to detect when it has traversed all the links in the network. The routing mechanism is used at each node to select the next link on which to send the process such that in a finite number of link traversals the process will detect termination.

The termination detection mechanism is implemented by a counter, called the *debt-counter*, which is carried by the process. The root starts the traversal by initializing the debt counter to zero and sending the process to itself. The counter is incremented by one as soon as the process arrives at a node for the first time (i.e., when the process is initially received by the root the counter is incremented to one). Then the counter value is tested; if its value is zero then the process is stopped at this node (see Fig. 1). Otherwise, the counter is decremented by one just before leaving a node through its last untraversed outgoing link. After leaving a node at least once through each of its outgoing links, the debt counter is never changed again at this node.

```
Response to receiving process($Debt\text{-}counter$) at node $v$
                    /* Also performed by the initiator with $Debt\text{-}counter = 0$ */
1.  If $v$ is unmarked
    then begin
2.      mark $v$
3.      $Debt\text{-}Counter := Debt\text{-}Counter + 1$ ;
    end
4.  else if $Debt\text{-}Counter = 0$ then stop;

5.  let $l$ be the next link in the cyclic order of $v$'s incident links.

6.  If $l$ is unmarked
    then begin
7.      mark $l$
8.      If $l$ is the last unmarked link of $v$
9.      then $Debt\text{-}Counter := Debt\text{-}Counter - 1$ ;
    end
10. Send the process($Debt\text{-}Counter$) over $l$
```

FIG. 1. *Traversal-1.*

LEMMA 2.1. *The debt-counter is equal to zero when it is tested* (*in line* 4 *in the code of Fig.* 1) *if and only if all links have been traversed.*

Proof. ($\leftarrow$) Clearly, for each newly visited node the process increments the counter by one. Similarly, for each visited node whose outgoing links have been all traversed, the process decrements the counter by one. Hence, when all links are traversed, the counter has been incremented and decremented the same number of times and is therefore back to its initial value.

($\rightarrow$) Assume that the counter is zero but not all the links in the network have been traversed. Since the network is strongly connected there is a directed path from a visited node to the tail node of any untraversed link. Let l be the first untraversed link on one of these paths. Thus, l is untraversed and its tail node v was visited. Hence, the counter was incremented at least once more (at v) than it was decremented, which leads to contradiction. □

The mechanism implied by Lemma 2.1 may serve as a termination detection mechanism. However, the process needs to employ a routing rule that will lead to termination. We present different routing rules, which lead to different traversal algorithms. These algorithms are presented in this and the next section.

The routing rule used by traversal-1 is as follows: every node orders its outgoing links cyclically, i.e., the first link in the order follows the last one. Each time that the process arrives at a node it is sent on the next outgoing link according to the cyclic order.

LEMMA 2.2. *Using the above routing rule the process eventually traverses all the links.*

Proof. Assume the contrary. Then, since the network is strongly connected and finite, there must exist a set S of nodes that have been visited infinitely many times. Thus by the routing rule each of the outgoing links of a node in S has been traversed infinitely many times.

Consequently, any node reachable from S has been visited infinitely many times. Since the network is strongly connected S is the whole set of nodes. Since each of the links outgoing from S has been traversed infinitely many times, and in particular each of the links has been traversed at least once, we have a contradiction. □

Lemmas 2.1 and 2.2 suggest the traversal algorithm of Fig. 1. Initially all nodes and links are assumed to be unmarked. To start the algorithm the root initiates a process with a debt-counter set to zero and sends the process to itself (i.e., it places the process in its input queue).

Figure 2 is an example of a network on which Traversal-1 requires 2^{n-1} link traversals. Let N_l be the number of traversals over link l in some execution of the algorithm. Clearly, $N_0 = 2 \cdot N_{1L} = 2 \cdot N_{1R}$ and $N_{iL} = 2 \cdot N_{(i+1)L} = 2 \cdot N_{(i+1)R}$ for $i = 1, \ldots, n-1$. Since N_{nL} and N_{nR} are both equal to 1 by the end of the traversal, $N_0 = 2^{n-1}$.

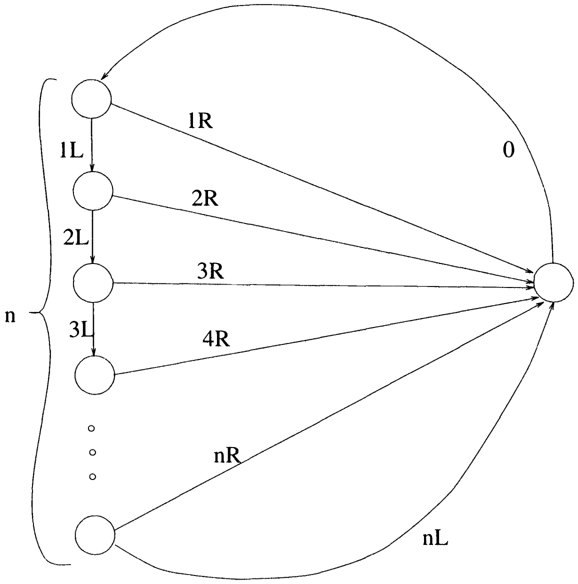


FIG. 2. *An example for the exponential complexity of Traversal-1.*

2.2. Traversal-2: Simulating directed depth first traversal. The source of Traversal-1's inefficiency is the routing procedure. A different routing rule is employed in this section to derive a traversal which requires $O(n \cdot m)$ link traversals.

We present the algorithm of Traversal-2 in three stages. First, a bidirectional depth first traversal algorithm is described. Second, a unidirectional implementation of the first algorithm is presented by assuming that a structure, called a *spanning in-directed tree*, is available. Finally, a mechanism to build the in-directed tree on the fly is given, thus providing a unidirectional traversal algorithm.

2.2.1. Bidirectional directed depth first traversal. Throughout the rest of the section we make a distinction between *unidirectional* and *directed* networks. A unidirectional network, as defined before, is a network in which some or all the links are unidirectional links. A directed network is a bidirectional network in which a unique direction is associated with each link. The link directions are given as part of the problem definition. Here we assume that the directed graph induced by the directions associated with the links is strongly connected.

In the bidirectional directed depth first search algorithm [HT73], the root spawns a process which visits all the nodes in the network. Upon arriving at node v for the first time, say through link l, the process marks v as *active*, l as the *parent link* of v, and iteratively traverses each of v's incident outgoing links. If the process arrives at an already marked node, it backtracks to the node from which it came. After backtracking on all of v's incident outgoing links, the process marks v as *fully backtracked* and backtracks from v on the parent incoming link, l. The traversal is completed when the root is marked fully backtracked.

A formal description of the algorithm is given in Fig. 3. Note that the process passed between the nodes is merely a token. It does not carry any information except its actual presence. Unlike this traversal, in the next sections we will use the process to carry control information between the nodes.

```
Initially all nodes and links are unmarked.
To start, the root $s$ performs:
    mark $s$ active ;
    select a link $l'$, outgoing from $s$ ;
    mark $l'$ active ;
    send the process over $l'$ ;

Response to receiving the process at node $v$ over incoming link $l$
If $v$ marked
then send the process back over $l$ ;
else
    mark $v$ active ;
    mark $l$ parent ;
    select a link, $l'$, outgoing from $v$ ;
    mark $l'$ active
    send the process over $l'$ ;
end

Response to receiving the process at node $v$ over outgoing link $l$
mark $l$ backtracked ;
If there is an unmarked outgoing link $l'$
then
    mark $l'$ active ;
    send the process over $l'$
else
    mark $v$ fully backtracked ;
    If there is no parent link
    then stop ;
    else send the process over the parent incoming link ;
end
```

FIG. 3. *The bidirectional directed depth-first traversal algorithm.*

The proof of the algorithm is a simple modification to the proof given in [Eve79]. The interested reader is referred to [Afe85] for the detailed proof. Here we state only the lemmas and corollaries.

LEMMA 2.3. *The bidirectional directed depth-first process traverses every link in the network at most once in each direction.*

COROLLARY 2.4. *The above traversal algorithm eventually terminates.*

LEMMA 2.5. *The bidirectional depth-first process traverses every link in the network once in each direction.*

COROLLARY 2.6. *The number of link traversals made by a traversing process in the bidirectional depth-first search is exactly* $2 \cdot m$.

2.2.2. Unidirectional depth first traversal, using a spanning in-directed tree.

The following two observations are used in this section to implement the bidirectional depth first traversal on a unidirectional network in which an in-directed spanning tree is defined. The resulting traversal makes $n \cdot m$ link traversals in the worst case.

Let the *parent node* of every node v, except the root, be the node from which the process arrived at v for the first time. At any given time, the link through which the traversing process left an active node most recently is called the *active link*.

Observation 1: The active nodes together with the active links form a directed path, called the *active path*. The first node on the active path is the root and the last link in the path either closes a cycle of active links (see Fig. 4), or leads to a node that is either fully backtracked or newly discovered. The most recently marked node among the active nodes is called the *focal point* of the traversal (e.g. see Fig. 4).

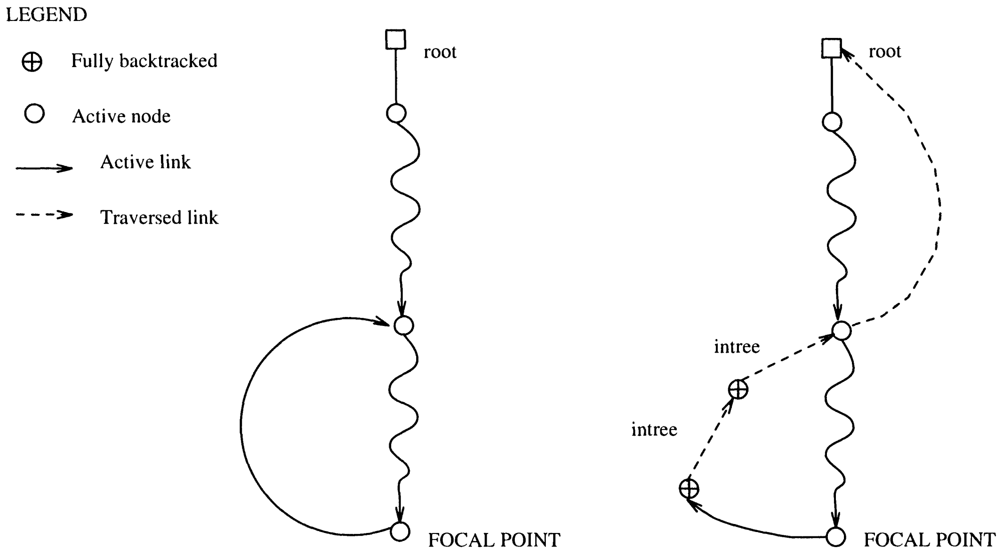


FIG. 4. *The active path.*

Observation 2: All backtracking is over the last link of the active path, i.e., either from an active node, or from a fully backtracked node to the last active node on the active path.

Observation 1 follows inductively from the fact that every node has at most one active outgoing link, and if it has one it must have an active incoming link (the parent link). Observation 2 follows immediately from Observation 1 and the algorithm.

An *in-directed tree* (or in-tree) is a subnetwork in which every node except one, called the *root*, has exactly one outgoing link, the underlying undirected graph is a tree, and all links are directed towards the root. An *in-directed spanning tree* is an in-tree which spans the network.

The difficulty in emulating the directed depth first traversal on a unidirectional network is that of emulating the backtrack, since links are unidirectional. In [Kob78], Kobayashi devised a solution to this problem with an exponential overhead, i.e., each backtracking incurred an exponential number of messages. To construct a linear overhead per backtracking, we note that whenever the process wants to backtrack over link l, a directed cycle, called the *backtracking cycle* is defined by concatenating l, the unique path in the in-tree from the head node of l to the root, and the active path. Thus, to backtrack over link l (from the head node of l to its tail node) the process goes along the backtracking cycle until it arrives at the tail node of l. To this end, the unique IDs of each node are used by the process to identify the tail node of l. Note that shortcuts are possible if the unique path of the in-tree intersects the active path before reaching the root (i.e., whenever the cycle is not simple). In particular, if the head node of l is active the process needs to follow only the active path in order to backtrack to l's tail node.

A formal description of the traversal algorithm is given in Fig. 5. The lines marked "I" are the code that the initiator has to execute in order to start the traversal.

To implement the backtracking mechanism, whenever the traversing process traverses link l from node v to node u it carries the ID of v. If u is unmarked (unvisited yet) then node u remembers that v is its parent. If node u is already marked, the process follows the cycle until it arrives back to v. When node u becomes fully backtracked the traversing process is sent along the backtracking cycle to u's parent.

LEMMA 2.7. *The number of link traversals made by a traversing process in the unidirectional depth-first traversal is a most $n \cdot m$.*

Proof. In Lemma 2.2 we saw that every link is backtracked exactly once. The lemma follows since in each such backtracking the process goes around a cycle of length at most n. □

In §2.5 it will be shown that $\Omega(n \cdot m)$ is also the lower bound on the number of link traversals.

2.2.3. On-the-fly in-tree construction. In this section we do away with the assumption that an in-tree is available by constructing the in-tree on the fly, while the process is traversing the network.

The essential use of the in-tree in the previous section was to backtrack from a fully backtracked node. Backtracking from an active node is accomplished simply by following the active path. While a node is active it computes its unique outgoing link in the in-tree, called *intree* as described below. While an active node may change its choice of the intree link, a fully backtracked may not. In this section we will maintain the following property throughout the algorithm execution.

Property 1: From any fully backtracked node, the unique path defined by following the intree marked links leads either to an active node, or if there are no active nodes, to the root.

The basic idea of the intree link selection is based on Property 1, above. We observe that the property is provided by maintaining a simpler property.

Property 2: From any fully backtracked node v the unique path defined by following the intree marked links leads to a node that was discovered before v (i.e., a node that would have a lower depth first number if such were assigned).

That is, at the time that v becomes fully backtracked (i.e., when its parent is still active) the intree links lead from v to some node u that is still active. The observation is that recursive applications of Property 2 ensure Property 1 (i.e., apply Property 2 to u, etc., as is formally stated and proved in Lemma 2.8).

Since the network is strongly connected, a path like the one required by Property 2 exists. Consider each backtracking cycle. Each such cycle consists of zero or more intree links and zero or more active links. If a backtracking cycle goes over an active link ($t \rightarrow h$), and it is the first time that a backtracking cycle goes over this active link, then from h's point of view, this cycle contains a path that leads from h to a node that was discovered before h was. Thus,

```
Response to receiving the traversal process, P, at node v:

If v is an unvisited node:
     v.parent := P.PreviousNode ;
I    v.activelink := any unused outgoing link ; {The "I" mark lines that are executed}
I    mark v active                                {by the root to start the algorithm}
I    P.PreviousNode := v.ID ;
I    send P over v.activelink ;

If v is an Active node and P is in the Forward mode:
     P.mode := backtrack ;
     P.FocalPoint := P.PreviousNode ;
     P.PreviousNode := v.ID ;
     send P over v.activelink ;

If v is an Active node and P is in the backtrack mode:
     if v.ID = P.FocalPoint
     then begin                          {v is the destination of the backtracking }
          if there are unused outgoing links
          then begin
               v.activelink := any unused outgoing link ;
               P.mode := Forward
          end
          else begin;                        {No more unused outgoing links : }
               mark v fully-backtracked ;
               if v has no parent then STOP;                    {v is the root }
               P.FocalPoint := v.parent ;
     end end
     P.PreviousNode := v.ID ;
     if v is still marked active
     then send P over v.activelink ;
     else send P over v.intree ;

If v is a Fully Backtracked node:
     if P.mode = Forward
     then begin
          P.mode := backtrack ;
          P.FocalPoint := P.PreviousNode ;
     end ;
     send P over the intree link ;
```

FIG. 5. *Traversal-2, the unidirectional depth-first traversal algorithm.*

the active links along the cycle, from h to the focal point, should be marked as intree links. Clearly such demarcation would not violate the property for descendants of h on the active path because this new intree path also provides them with Property 2.

To implement the above idea every node remembers whether or not its parent incoming link has already participated in a backtracking cycle. When the parent incoming link of node v participates in a backtracking cycle for the first time all the active nodes from v to the end of the active path select their present active link as their intree link. In the rest of this section a parent link which has never participated in a backtracking cycle is called a *bridge*.

Aside from the in-tree construction, the traversal is the same as the depth first traversal given in the previous subsection. It is assumed here that whenever a shortcut in the backtracking cycle is possible it is done, i.e., the backtracking cycle is a simple directed cycle.

The mechanism to construct the in-tree can be viewed as an approximation of the mechanism to determine the low-points of vertices in a directed graph, which was introduced by Hopcroft and Tarjan [HT73], [Eve79] in their algorithm for strongly connected components.

A formal description of the traversal algorithm, with the in-tree construction, is obtained by adding the code in Fig. 6 to the code in Fig. 5. The codes are combined in the following way. In case the traversal process arrives at an active node in the backtracking mode, then first the corresponding code in Fig. 6 is executed, and only then is the corresponding code of Fig. 5 executed. The lines that are marked by an "I" are the steps which the root executes in order to start the traversal.

```
Response to receiving the traversal process, P, at node ν:

If ν is an unvisited node:
        ν.BrdgHd := true;  {BrdgHd since ν's parent link have not yet been on a cycle }
        Continue with the code of Fig. 5 for the same case (unvisited ν)

If ν is an Active node and P is in the backtrack mode:
        if (ν.BrdgHd) and (P.PreviousNode =ν.parent)
        then begin              {1-st time that ν's parent link is on a backtracking cycle}
                P.XBrdg := true ;
                ν.BrdgHd := false ;
        end
        if P.XBrdg then ν.intree := ν.activelink ;
        if ν.ID = P.FocalPoint
        then P.XBrdg := false ;              {ν is the destination of the backtracking }
        Continue with the code of Fig. 5 for the same case (Active ν and
                                                         P in the backtrack mode)
```

FIG. 6. *Traversal-2, the additional code for the in-tree construction.*

In this algorithm node ν knows which of its incoming links is the parent link by recording the ID of the node on the other side of the parent link; this is the parent node of ν. The first time that a parent incoming link of node ν participates in a backtracking cycle is easily detected, because it is exactly the second time that the process arrives at ν through this link. To this end, a boolean variable, called BrdgHd (Bridge Head), is used at every node to indicate whether its parent incoming link is a bridge (i.e., if it has already participated in a backtracking cycle). Another boolean variable, called XBrdg (crossed bridge), is used on the traversing process to indicate whether the current backtracking cycle goes over a bridge. Whenever the process arrives at an active node ν in the backtracking mode, and the XBrdg indicator is on, ν selects its active link to be its intree link. Aside from BrdgHd, every node ν has the following fields: *ID*, which is the ID of ν; *parent*, which is the ID of the parent node of ν; *activelink*, which points to the activelink of ν; and *intree*, which points to the intree outgoing link of ν. Aside from XBrdg, the traversing process p, has the following fields: *mode*, which indicates whether the process is now backtracking or not; *PreviousNode*, which is the ID of the node that p visited last; and *FocalPoint*, which is used in the backtracking mode and is the ID of the backtracking destination node.

It remains to prove that the intree links selected by any fully backtracked node span all these nodes and always lead to an active node.

Let us define an *in-directed forest* as a collection of disjoint in-trees.

LEMMA 2.8. *The intree marked links of the fully backtracked nodes constitute an in-directed forest.*

Before proving the lemma we note the following two implications of its proposition. First, if there are still active nodes, then the roots of in-trees in the forest must be active nodes, which is exactly what we need for the backtracking process. This is because all the fully backtracked nodes have an intree outgoing link which obviously cannot lead to an unvisited node. Second, the proposition implies that when the algorithm terminates, the intree links constitute a spanning in-directed tree rooted at the traversal initiator, the root.

Proof of Lemma 2.8. The claim will be proved by induction. Clearly, the lemma holds when the algorithm starts at a time when no node is fully backtracked. It is also clear that the lemma holds when the first node to become fully backtracked, becomes fully backtracked, since one node with one outgoing link make up a legal tree. Assume that the claim holds just before node v becomes fully backtracked, and we will prove that it holds after v becomes fully backtracked.

If v is the root then the claim certainly holds, since v selects no in-tree link. Henceforth assume v is not the root, and when v becomes fully backtracked there is at least one active node in the network (the root).

Assume to the contrary that after v becomes fully backtracked the claim does not hold. Let B be the set of nodes which were already visited before v was visited for the first time. Let A be the set of nodes that includes v and all the nodes that were visited for the first time after v was, and before v became fully backtracked. Let L be the set of links which are directed from a node at A to a node at B. Clearly, $L \neq \emptyset$ since the network is strongly connected. By the definition of A and B there is no traversed link from B to A except the parent link of v, before v becomes fully backtracked. Thus, all the in-trees in B are rooted at active nodes in B and each backtracking cycle, C_l, which is associated with each $l \in L$ passes from B to A on the parent link of v. Clearly, at least one $C_{l,}$ $l \in L$, passed on a bridge (the parent link of v was a bridge when v was visited for the first time). Let link $l^* = (a \rightarrow b)$ be that link in L whose associated backtracking cycle, C_{l^*}, was the last cycle among the backtracking cycles associated with the links in L to pass over a bridge before v becomes fully backtracked.

Cycle C_{l^*} is used as a backtracking cycle for the first time when l^* is traversed for the first time, which is when a is still active. Since at this time C_{l^*} passes over a bridge, all the links from v to b on C_{l^*} must have been marked as intree links. Since no node in B becomes fully backtracked after v is visited and before v becomes fully backtracked, no intree mark of a node in B is changed at that period. Also, since C_{l^*} is the last backtracking cycle to pass over a bridge before v becomes fully backtracked, all the intree marks along C_{l^*} do not change before v becomes fully backtracked. Hence, the intree links that have been selected by v when v becomes fully backtracked cannot close a cycle of fully backtracked nodes and intree marked links, which is a contradiction. □

2.2.4. Reducing the communication complexity of Traversal-2 to O ($n \cdot m + n^2 \log n$) bits. In this section Traversal-2 is modified so in at most $2n - 2$ of its backtracking cycles the process will carry $O(\log\ n)$ bits around the backtracking cycle. In the rest of the backtrackings the process need not carry more than a constant number of bits. Thus the bit complexity is reduced from $O(n \cdot m \cdot \log\ n)$ to $O(n \cdot m + n^2 \cdot \log\ n)$. A variation of the traversal presented here is also presented in §4 in a much different setting.

The modification of the algorithm is as follows:

1. Every active node uses a boolean variable, called the *focal point*, to assert whether or not it is the focal point of the traversal. If the focal point variable of node v is false then v is not the focal point of the traversal. When v is visited for the first time it sets its focal point to true. When v becomes fully backtracked it sets its focal point to false.
2. Whenever the process arrives in the forward mode at an already marked node (i.e., an active node), v, a two-phase backtracking is started. In the first phase the process is sent around the backtracking cycle and back to v, counting whether there are one or more nodes on the backtracking cycle whose focal point is true. To this end only a constant number of bits has to be carried around by the process.
3. If only one node on the cycle has its focal point "on," then this node must be the node preceding v on the cycle (see Fig. 7, case 1), i.e., it is the backtracking destination. In this case, the process is sent around the backtracking cycle again with a constant number of bits, to the unique node which asserts itself as the focal point. Hence, a complete backtracking is performed with the process carrying only a constant number of bits.

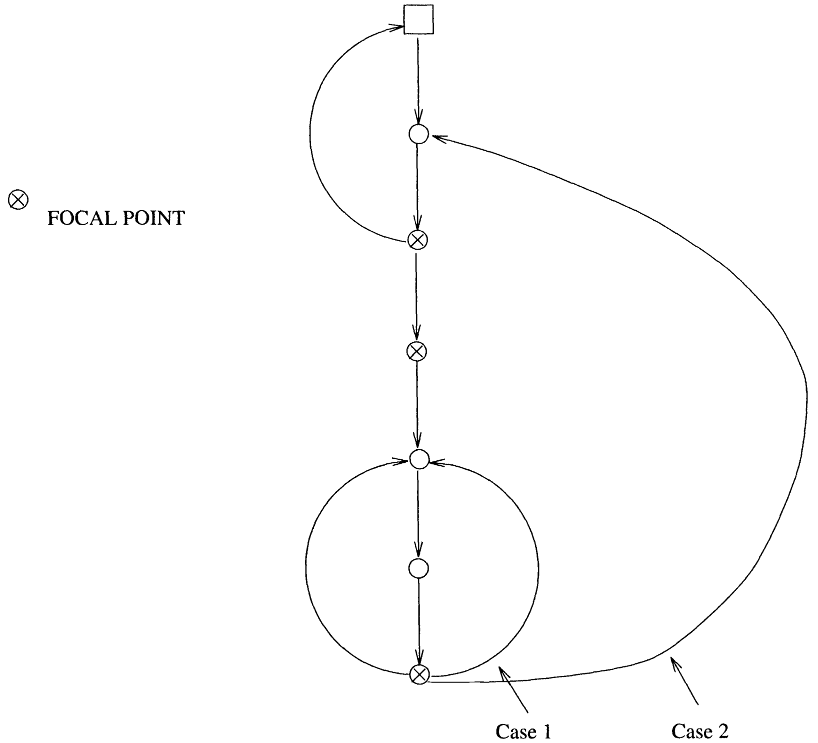


FIG. 7. *Backtracking in Traversal-3.*

4. If more than one node on the backtracking cycle has its focal point "on," then a bridge was included in the backtracking cycle (see Fig. 7, case 2) and a backtracking identical to the one used in Traversal-2 is initiated by v (with XBrdg set to true). In this phase of the backtracking all the nodes, aside from the last one on the active path, set their focal point to false. A bridge is included since the active link leaving any node whose focal point is true, aside from the last one, must have led to a new node

(i.e., it is the parent link of the next node on the active path) and has never been on a backtracking cycle before (otherwise the focal point would not have been true).

5. The backtracking from a fully backtracked node remains the same as in Traversal-2 except that the focal point of the destination is set to true.

The main claim of this subsection is given in the following lemma.

LEMMA 2.9. *By the above modification the process will have to carry $O(\log n)$ bits around a backtracking cycle only $2n - 2$ times.*

Proof. The process has to carry $O(\log\ n)$ bits around the backtracking cycle either when it backtracks from a fully backtracked node to its parent, or when the backtracking cycle goes over some bridge for the first time. Thus we can associate one such backtracking with each node that becomes fully backtracked, and one with each bridge. Clearly there are $n - 1$ nodes which become fully backtracked (except the root from which the process never backtracks). Similarly, there are $n - 1$ bridge links since each such link is the unique incoming parent link of some node (except the root which has no bridge link entering it). □

In the remaining $m - 2n + 2$ backtrackings the process carries only a constant number of bits. Since every backtracking requires at most n link traversals we get the following corollary.

COROLLARY 2.10. *The communication complexity of the modified Traversal-2 is $O(n \cdot m + n^2 \cdot \log n)$ bits.*

2.3. Traversal-3: An algorithm for a network of finite automata. In this section the assumption that every node has $O(\log\ n)$ bits of memory is relaxed. Instead, every node is assumed to be a finite automaton, i.e., to have constant size memory regardless of the network size, in which case the nodes do not have unique IDs. Since each node has a constant number of memory bits, the traversing process has to be of constant size too. We will show that with a constant size process the traversal requires at most $O(n \cdot m + n^2 \cdot \log\ n)$ link traversals, which is also the bit complexity of the algorithm. Note that the only nonconstant space used by Traversal-2 is the usage of $O(\log n)$ bits to carry the identity of the backtracking destination (this is also the only usage of the unique IDs) in each of the backtrackings considered in Lemma 2.9. Here we show how each of these backtrackings can be implemented by a constant size process which will go around the backtracking cycle $O(\log\ n)$ times.

To recognize the preceding node on the backtracking cycle using a constant size process, we use a solution to the following "last in the ring" puzzle: In a unidirectional ring of finite automata, design an algorithm by which a designated node, v, will distinguish the node preceding it, u, from all other nodes.

LEMMA 2.11. *The bit communication complexity of the "last in the ring" puzzle in an asynchronous ring is $\Theta(n \cdot \log\ n)$.*

Proof. A solution to the puzzle works in $\log n$ phases. Initially, all nodes except v are candidates for the position. In each phase we eliminate half of the remaining candidates by sending a token around the ring, alternately marking the candidate nodes even and odd. When the token arrives at v, it remembers the parity of the candidate preceding v. In the next phase, the token eliminates all candidates whose parity differs from the parity of the desired node. The last phase is detected by the token when it sees that only one node has not been eliminated. Thus, the token carries one more bit to indicate whether there are one or more nodes that have not yet been eliminated on the cycle. Hence, $O(n \log\ n)$ is an upper bound on the bit complexity of the puzzle.

To prove that this is also the lower bound we first claim that the sequence of bits transmitted over each link must be unique. Otherwise, there must be two distinct links, $u_0 \to u_1$ and $v_0 \to v_1$, such that both have seen the same sequence. Since the network is asynchronous, u_1 and v_1 are in the same state and both have generated the same sequences on their outgoing links, which implies that their down neighbors u_2 and v_2 are also in the same state. Continuing this argument inductively, we conclude that the node preceding the designated one has to be at an equal distance from both u_0 and v_0. This is a contradiction; hence, the sequence of bits transmitted over each link is unique. Since with $\lfloor \log(n) \rfloor - 1$ bits at most $n/2$ unique sequences can be written, over at least $n/2$ of the links $\Omega(\log n)$ bits were transmitted. Thus, $\Omega(n \log n)$ is also the lower bound. □

Thus, whenever node v becomes fully backtracked, or a backtracking cycle with more than one focal point is closed at v, v starts the algorithm described in the proof of Lemma 2.11 to send the process to the node preceding it. The total bit complexity of the traversal does not change with this modification; however, the number of link traversals made by the process is linear with the bit complexity of the traversal, i.e., $O(n \cdot m + n^2 \cdot \log n)$.

2.4. Traversal-4: An algorithm for a synchronous network of finite automata. Traversal-3 can be run on a synchronous network without any modification, in which case it will exhibit the same performances. The question arises whether any gains can be achieved in the synchronous model. To answer that question positively, we provide an $O(n \cdot m)$ time and bit complexities traversal algorithm for an arbitrary topology, unidirectional synchronous network, which matches our general case lower bound.

Following the proof of Lemma 2.9 and in the same spirit as §2.3, we show here how each backtracking from Lemma 2.9 can be implemented by a constant size process which goes around the backtracking cycle $O(1)$ times.

LEMMA 2.12. *The bit complexity of the "last in the ring" puzzle in a* synchronous *ring is* $\Theta(n)$.

Proof. $\Omega(n)$ is clearly a lower bound since a signal has to go around the ring from the designated node to the node preceding it.

An algorithm to match this lower bound uses two tokens that start moving around the ring from the designated node. The synchronous mode of communication is used to move one token at half the speed of the other token. The fast token makes one hop in each round of the computation, while the slow token makes one hop every two rounds. Both tokens are moving for $2n - 1$ rounds, after which the node preceding the designated node removes them from the ring. The fast token catches up with the slow token at the node preceding the designated node after going around two times. The node preceding the designated node is the first node to receive the fast token for the second time while holding the slow token (each node holds the slow token one extra round). To see this let f_t and s_t be the indices of the nodes holding the fast and the slow tokens, respectively, at round t. Clearly $f_t = t \bmod n$ and $s_t = \lceil \frac{t}{2} \rceil \bmod n$. The result follows since $n - 1$ is the lowest positive value of t at which $f_t = s_t$. □

2.5. Lower bounds. After arriving at the upper bounds of $O(n \cdot m + n^2 \cdot \log n)$ bits and $O(nm)$ bits on the communication complexity for traversal, we determine the lower bound. Much research is still needed on establishing tight lower bounds on the unidirectional traversal problem in general. In this section we present one step in this direction. We show that $\Omega(n \cdot m)$ is a lower bound on the number of link traversals required by a single token traversal, i.e., when the traversal is restricted to send at most one message at a time.

LEMMA 2.13. $\Omega(n \cdot m)$ *bits is the lower bound to a single token traversal of a unidirectional graph of arbitrary topology.*

Proof. The proof is by example; see Fig. 8. The result follows since each traversal of a link from A to B must be followed by a traversal of the path C. □

Lemma 2.13 proves that our traversal algorithm is optimal for dense networks (in which $m = \Omega(n \cdot \log n)$) and under the restriction that the traversal has at most one outstanding message at a time. Furthermore, it suggests that the algorithm is optimal in the general case.

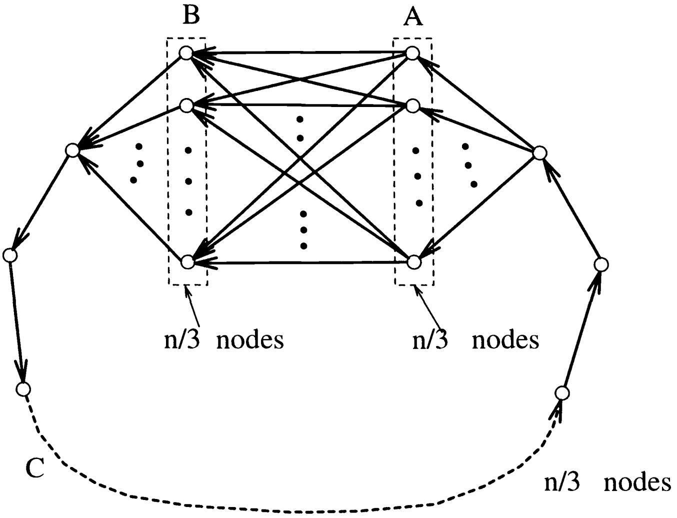


FIG. 8. *A network for the $\Omega(n \cdot m)$ lower bound.*

2.6. Producing a spanning out-tree. Traversals 2, 3, and 4, which are different implementations of the same algorithm, can be modified to produce a useful structure, called an *infrastructure*, on the network. The infrastructure is the combination of an in-directed spanning tree and an out-directed spanning tree. The in-tree construction was detailed in §2.2.3. Here we will describe how an out-tree may be produced by Traversal-2. Note that the defined infrastructure is a strongly connected subnetwork which spans the network and has $2n - 2$ links. The infrastructure proves to have several practical applications, outlined in §4.3.

An *out-directed tree* (or out-tree) is a subnetwork in which every node except one, called the *root*, has exactly one incoming link and the underlying undirected graph is a tree. Since every node in the out-tree has exactly one incoming link, there is a unique path from the root to every node in the out-tree. An *out-directed spanning tree* is an out-tree which spans the network. An example of an out-tree is given in Fig. 9.

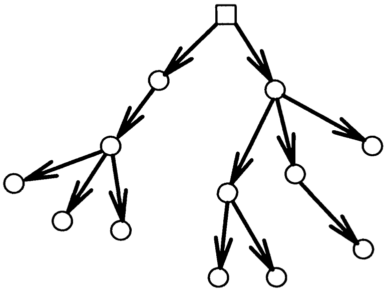


FIG. 9. *An example of an out-tree.*

We now explain how every node in the network marks some of its outgoing links *outtree* during the traversal algorithm such that, when the algorithm terminates, the collection of outtree marked links constitutes a spanning out-tree of the network rooted at the traversal initiating node.

To construct an out-tree we make the following observation: the collection of parent links constitutes an out-tree. To prove it note the following: (1) every node, except the root, has exactly one parent incoming link, and (2) going backward on the path defined by the parent links from any node v, we always arrive at the root. Note that (2) follows from the fact that the incoming parent link of any node v connects v to a node which was explored before v, thus we cannot close a cycle and we must arrive at the root.

To detect the parent outgoing links of node v we observe the following: the traversing process leaves node v twice or more through link l (in Traversal-2) while v is *active* if and only if l is a parent link. Thus, every *active* node counts whether or not each of its outgoing links has been on a backtracking cycle more than once. If a link participated in a backtracking cycle more than once while v is active then it is a parent link and is marked outtree.

3. Election in unidirectional networks.

In this section we present a recursive distributed algorithm for election in unidirectional strongly connected networks. One version of Traversal-2 (where an intree is assumed to exist in the network, see §2.2.2) is used as a tool in the construction of the election algorithm, while Traversals 2 and 3 with no a priori structures are a special case of the election algorithm, as shown in the next section.

3.1. Definitions and outline.

The election algorithm is based on the following recursive properties of strongly connected directed multigraphs:

1. The set of links, defined by selecting one outgoing link from every node, contains a nonempty set of disjoint directed cycles.
2. The subgraph, obtained from G by contracting any of the cycles defined above into one node, results in a strongly connected multigraph.
3. Repeated application of the operation in 1 and 2 contracts G into a single node.

The distributed algorithm proceeds in conceptual phases, which follow the above contraction process. When a cycle is detected, its nodes are grouped into a cluster. Similar phases are used in [Hum83]. Initially we consider each node to be a single node cluster. The phases of the algorithm are selection of an outgoing link from each cluster, called a *selected* link, detection of cycles among clusters, and contraction of cycles of clusters.

A cluster is recursively defined as follows:

1. A single node is a cluster.
2. A set of clusters that are joined in a ring by their selected outgoing links is a cluster.

Recursively we assume that every cluster satisfies the following four properties (see Fig. 10):

1. A unique node in the cluster is distinguished as the *cluster-head.*
2. All the nodes in the cluster know the ID of the cluster-head, which is also the *ID of the cluster.*
3. Each node in the cluster, except the cluster-head, has one outgoing link marked as *intree* link. The collection of intree links forms a directed incoming tree, spanning the cluster and rooted at the cluster-head.
4. A strongly connected subnetwork which spans the cluster, called the *infrastructure* of the cluster, is defined on the cluster.

The cluster head of the unique final cluster that contains the whole network is the elected leader. Clearly, a single node cluster satisfies the inductive assumptions. It is the cluster-head of itself, a single node in-tree, and a strongly connected subnetwork. To describe the algorithm we will describe the inductive step, i.e., we assume that a set of clusters which satisfy the assumptions already exists and explain how a bigger cluster which satisfies the inductive assumptions is composed out of this set.

To select a cluster outgoing link, each cluster head initiates a depth-first traversal (DFT) process (Traversal-2). The traversal process is used to search for a link which is potentially outgoing from the cluster. For the depth-first traversal algorithm we use the traversal algorithm developed in §2.2.2.

To detect a cycle, we use a simple algorithm for election on a unidirectional ring. Each cluster forwards the largest ID it has seen. When a cluster receives the same ID twice it has detected a cycle.

The contraction of a cycle is accomplished by first electing one of the cluster-heads on the cycle to be the cluster-head of the expanded cluster. Then, the newly elected cluster-head synchronizes the cluster by broadcasting the new cluster-ID to all the nodes and constructing all the inductive requirements on the new cluster.

The most expensive phase is the DFT in a search for a cluster outgoing link, because in the contraction phase we lose the DFT information accumulated by all the clusters around the cycle except one. When the cluster-head initiates a DFT in the next phase, the search will have to spend much effort regaining all the lost DFT information. However, by selecting the cluster-head of the largest cluster on the ring to be the new cluster-head, we minimize the amount of information lost. Thus, we limit the rate of information loss to the rate of cluster growth (i.e., if a large cluster were contracted with a small one, the amount of information lost is proportional to the size of the small one). In fact, we are able to show that the cost of all the DFTs conducted during the algorithm is within a constant factor of the cost of a single DFT. Since this point is critical to the complexity calculation, but rather minor to the description of the algorithm, we postpone a detailed discussion of it until the complexity section.

After a cluster is formed, its nodes are synchronized to search for an untraversed link outgoing from the cluster. To achieve this synchronization, the in-tree rooted at the cluster-head is used. When a cycle of clusters is contracted into a bigger cluster, all their in-trees are merged into one in-tree, spanning the new cluster. The operations of merging in-trees and searching clusters utilize each other alternately. The structures left by the DFTs are used to modify and merge the separate in-trees around the cycle into one in-tree. In turn, the in-tree in a cluster is used for routing purposes, by its DFT process (see §2.2.2.).

In the next three subsections we present the three phases of the algorithm, starting with the cluster outgoing link selection (see Fig. 10). During the algorithm, links are in one of three states: *new*, *elementary*, or *killed*. A *new* outgoing link is one which has not yet been traversed. An *elementary* link is a link which was a cluster selected outgoing link during one of the previous stages. The set of elementary links within one cluster forms the infrastructure of the cluster. A *killed* link is an already traversed link that has never been selected as the outgoing link of a cluster (i.e., an intra-cluster link that is not elementary).

3.2. Selection of a cluster—outgoing link. Once a cluster is formed, its head node initiates a DFT algorithm to search the cluster's infrastructure for a node with an untraversed outgoing link. The first such link to be found is selected as the cluster's outgoing link. If it turns out to be an intra-cluster link, the DFT continues where it was stopped in the search of another untraversed outgoing link. If no cluster outgoing link is found, the cluster contains all the nodes of the network and the algorithm terminates.

The cluster outgoing link selection phase begins after the synchronization of the cluster has terminated. The head node of the new cluster initiates a DFT token on the cluster's infrastructure. The token carries the ID of the cluster which created it and the maximum cluster-ID observed so far (maximum-ID). The cluster-ID is used to distinguish between inter- and intra-cluster links, and the maximum-ID is used to detect a cycle of clusters. The DFT token searches the cluster for a new link, i.e., a link that has never been used by the algorithm. In doing so, the DFT token leaves behind a trail of active links, which leads from the cluster-head to the token location (which is the traversal focal point).

Upon finding a new outgoing link, l, the token traverses link l to node v on the other end of l. If v belongs to the same cluster, the token returns to l's tail via the in-tree and the active path. Link l is then marked killed, and it will never again be traversed. If, on the other hand,

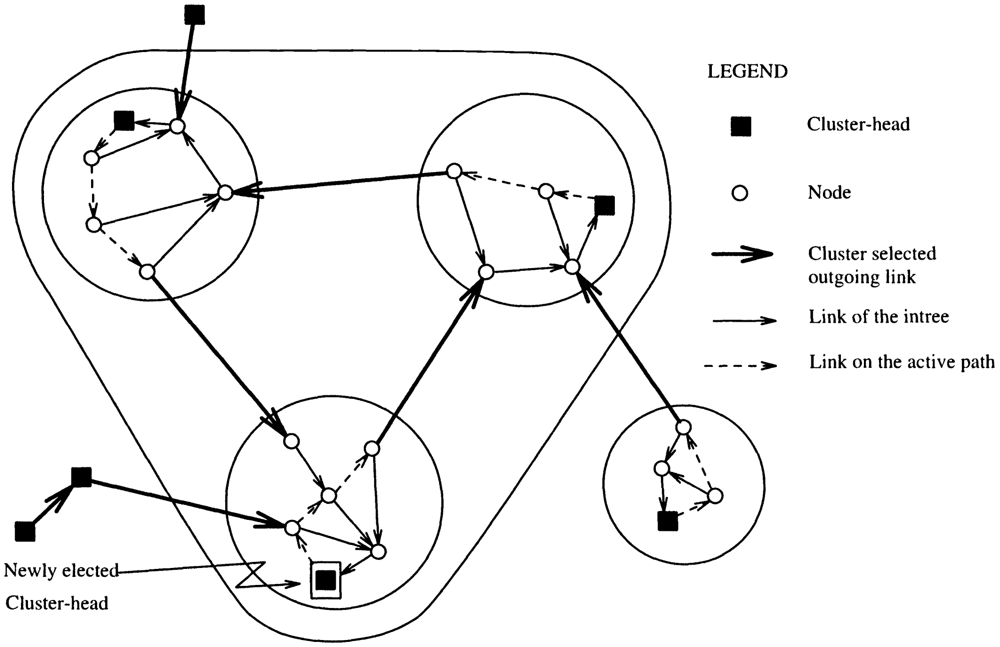


FIG. 10. *Clusters in the election algorithm.*

the token arrives at a different cluster, the information it carries and the newly selected link it has established enable the cycle detection to continue as described below.

3.3. Cycle detection. For the sake of simplicity, we have selected an inefficient algorithm for cycle detection. Since its complexity is, in general, not the bottleneck, we have avoided discussing an improved mechanism for cycle detection (the improvements are a generalization of [Pet82], [DKR82], with which our algorithm will perform optimally on rings).

After each cluster selects an outgoing link, the network contains at least one cycle which consists of two or more clusters (see Fig. 10). Let each cluster send its ID on the cluster outgoing link. A cluster forwards another cluster-ID only if it is larger than all the cluster-IDs it has received in the past. Eventually, one and only one cluster in each cycle will receive the same cluster-ID twice, thus detecting a cycle. The cluster-head that detected the cycle synchronizes the new cluster.

The operation of cycle detection is carried out by the cluster-heads. To implement it, each node receiving a message from a different cluster forwards it to its cluster-head through the cluster's in-tree. To forward a maximum-ID from a cluster-head to the next cluster, the cluster-head broadcasts the maximum-ID over the infrastructure of its cluster. All nodes (including the cluster-head) retain a maximum-ID variable, which is updated by the maximum-ID of the broadcast message. The DFT token updates its maximum-ID to the largest it encounters along its way. If a cluster outgoing link has been selected, the broadcast message is simply forwarded over the outgoing link to the next cluster.

When a cluster-head receives a maximum-ID which is equal to its own, it has detected a cycle and it is elected to start the synchronization phase.

3.4. Cycle contraction and cluster synchronization. In the cycle detection phase a new cluster with an elected cluster-head was found. In this phase the elected cluster-head establishes the remaining three inductive assumptions on the new cluster. The elected cluster-head thus (1) notifies all the nodes in the clusters around the cycle of their new cluster-ID,

(2) combines the in-trees of all the clusters into one in-tree which spans the new cluster and is rooted at the elected cluster-head, and (3) combines the infrastructures into one infrastructure spanning the new cluster. After receiving a positive acknowledgment that all the nodes have completed these constructions, the elected cluster-head starts the next phase of cluster outgoing link selection. The synchronization phase is carried out by a broadcast with echo mechanism on the cluster new infrastructure.

Upon receiving the first copy of the broadcast message, every node performs the following five operations in the following order: (1) It updates its own cluster-ID. (2) If one of its links is a selected cluster-outgoing link then it marks that link as elementary and sends a message to this effect over that link. Every node maintains a list of its incoming elementary links, which it updates on the reception of such messages (this list is necessary for the echo phase of the broadcast and echo described below). (3) It updates (if necessary) its intree mark. (4) It forwards copies of the broadcast message over its elementary outgoing links. (5) It acknowledges the reception of the message through the in-tree. Any duplicate copies of the message are discarded.

The second operation combines the infrastructures into one infrastructure which spans the new cluster.

Merging the in-trees. To merge all the in-trees around the cycle, we use the active paths in each cluster which lead from the cluster-heads to the head nodes of the selected outgoing links. The active paths are constructed during the DFT in the cluster's outgoing link selection phase (see §2.2.2).

We notice that if each node of a cluster that has an active outgoing link puts its intree mark on the active link, then the incoming spanning tree is rerooted to the *head* node of the cluster outgoing link (see Fig. 11). (Note that the head node of the cluster outgoing link belongs to the next cluster on the cycle of clusters.) To obtain a directed in-tree which spans all the clusters around the cycle, we perform this operation in all the clusters around the cycle except for the elected cluster. Thus, the in-tree formed is rooted at the elected cluster-head. This in-tree is used by the nodes to notify their new cluster-head of the contraction termination.

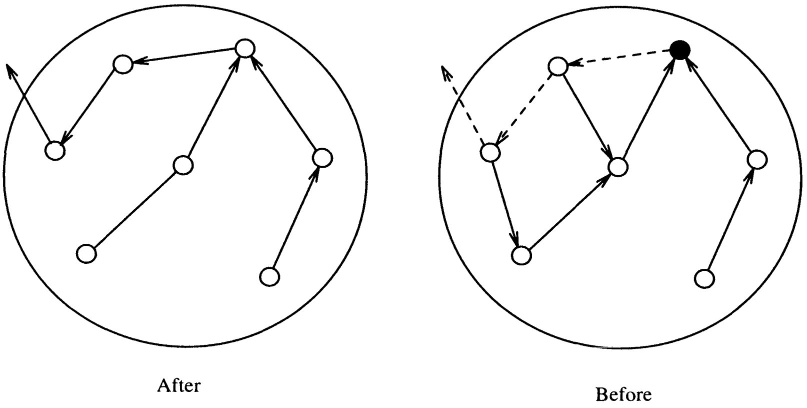


FIG. 11. *Rerooting an in-tree.*

Acknowledging the broadcast. To notify the cluster-head that all the nodes in the new cluster are aware of their new cluster-ID, every node sends an acknowledgment as follows. After receiving the first copy and making all the necessary updates, every node sends an acknowledgment over all the elementary outgoing links aside from the intree marked link. An acknowledgment is sent over the intree marked outgoing link only after an acknowledgment has been received on all the elementary incoming links.

When the elected cluster-head has received the acknowledgment message over all of its elementary incoming links, the contraction has terminated, and a new phase of cluster outgoing link selection is started.

To prevent messages with new and larger maximum-IDs from being lost during the contraction phase we specifically forward them as follows. First, every node that receives a new and larger ID forwards it to its cluster-head in a special message. Second, when a cluster-head broadcasts a new and larger maximum-ID it also does it in a specifically marked message. These messages are not destroyed by the contraction operations unless they encounter a node with a larger maximum-ID. Moreover, any new and larger maximum-ID which arrives at the cluster-head that has detected the cycle (in such a special message) during the synchronization phase is held back by this cluster-head. It is redelivered to the cluster-head upon termination of the synchronization broadcast and echo.

Termination. The algorithm terminates when a cluster fails to select a cluster-outgoing link. This cluster spans the whole network; its cluster-head is the elected node and its maximum-ID is the largest ID.

3.5. Complexity of the election algorithm. Communication complexity analysis involves counting the total number of bits transmitted over all the network links. The communication cost of the algorithm has three components: the cost of cluster synchronization, the cost of cycle detections, and the cost of cluster-outgoing link selection.

We observe the following two facts.

Fact 1: The number of cycle detections is at most $n - 1$.

Fact 2: In a cluster of size k, there are at most $2k - 1$ *elementary* links.

Fact 1 holds because the contraction of a cycle strictly reduces the network size. Fact 2 follows from Fact 1 and the observation that there exists a one-to-one correspondence between clusters and elementary links.

LEMMA 3.1. *The total cost of synchronizing the clusters is at most $O(n^2 \log n)$ bits.*

Proof. According to Fact 1 there are at most $n-1$ cluster synchronizations. The synchronization messages propagate on the infrastructure of a cluster which contains at most $2n - 1$ links. In each synchronization one broadcast message and one echo message are transmitted on each elementary link. Since each message is of size $O(\log n)$ bits the result follows. □

LEMMA 3.2. *The total cost of all cycle detections is at most $O(n^2 \log n)$ bits.*

Proof. Each time a new cluster-head is elected a new maximum-ID is sent around a cycle of clusters. Thus, by Fact 2, there are at most $2n - 2$ maximum-ID initiations. Each such initiation is sent over the infrastructure of some cluster. The same maximum-ID is forwarded at most three times on the same link in a particular cluster: once when it enters the cluster and the link is an in-tree link on which it was forwarded to the cluster-head; once when the cluster-head broadcasts the maximum-ID; and once on its return to that cluster-head (which then detects a cycle). Since a link forwards the same maximum-ID at most three times for each cluster it belongs to, the result follows. □

The cost involved in selecting outgoing links consists of the cost of killing inter-cluster links and the cost of the DFTs.

To kill link l, the algorithm transmits one message of $O(\log n)$ bits over l and a kill message of size $O(1)$ bits over a path from the head of l to its tail. The kill message goes down the in-tree to the cluster-head and then along the active path to the tail of l. This node is distinguished from other nodes on the active path since it is the focal point of the DFT. Thus, the killing of one link costs $O(n)$ bits. Since the algorithm kills, at most, m links, the killing of intercluster links adds up to, at most, $O(n \cdot m)$ bits.

As mentioned, the cost of one DFT on a network with n nodes and m links is $O(n \cdot m \cdot \log n)$ since there is one backtracking for each link of the network and each backtracking costs $O(n \log n)$. The DFT operation is employed in the election algorithm to search the infrastructures of different clusters. Since there are, at most, twice as many links as nodes in an infrastructure, the cost incurred by each DFT of a cluster with k nodes is $O(k^2 \log n)$. As the DFT could be used $n - 1$ times by the algorithm, the total cost of all DFTs might be $O(n^3 \log n)$.

To reduce the total cost of all DFTs from $O(n^3 \log n)$ bits, a special *cluster-head election* phase is added after the cycle detection and before the synchronization phases. In this phase the cluster-head which is elected in the cycle detection phase synchronizes the election of the cluster-head of the largest size cluster around the cycle. The new cluster-head then proceeds with the cluster synchronization phase as described before. After synchronizing the cluster, the cluster-head resumes its DFT process from the previous stage, i.e., from the head node of its former cluster outgoing link, thus avoiding re-searching nodes that were already searched. The head node of its former cluster outgoing link is now the last node on the active path.

Cluster-head election. To elect a cluster-head of a largest size cluster (ties are resolved by cluster-IDs), the cluster-head detecting the cycle sends an elect-message around the alternating sequence of active paths and in-trees which form a directed cycle (see Fig. 10). On its way, the elect-message finds out which cluster has the largest number of nodes and what the total number of nodes in all the clusters around the cycle is. This information is updated on the elect-message by the cluster-heads along the directed cycle.

Once the elect message returns to the originating cluster-head, the control of cluster synchronization is passed to the newly elected cluster-head. Any new and larger maximum-ID which arrives at the cluster-head detecting the cycle during the election and the synchronization phase is held back by this node. It is delivered to the new cluster-head upon the reception of the synchronization broadcast.

After being elected, the new cluster-head resumes its DFT of the previous cluster outgoing link selection phase. In doing so it uses the active path and the node marks which its DFT had left. The algorithm thus can be viewed as a process in which all the cluster-heads are candidates for leadership. When clusters owned by different candidates form a cycle, the candidate which owns the largest size cluster eliminates all the other candidates. In doing so the candidate merely has enlarged its cluster to include the clusters of the candidates it has eliminated. The DFT structures it had in its original cluster are now extended to search the enlarged cluster. Henceforth, we refer to the clusters which did not change their cluster-ID as one cluster which has consumed other clusters during the algorithm.

The above scheme is similar in principle to the capturing rule of [Gal77] and algorithm A in [AG91]. There we enabled the largest candidate, in terms of captured nodes, to kill and take nodes from a smaller candidate. To see that the above scheme does not add more than a constant factor to the complexity of a single DFT, we will use a lemma which was introduced in a similar context by Gallager [Gal77].

LEMMA 3.3. *For any given k, the number of clusters that own n/k nodes or more is, at most, k.*

Proof. Let C_1 and C_2 be any two clusters which had size n/k at some point of time. We shall show that each of C_1 and C_2 must have had n/k nodes disjointly. If they have never consumed each other, we are done, since the clusters were certainly disjoint. If, w.l.o.g., C_1 has consumed C_2, then C_1 must have already had n/k nodes at the time of eliminating C_2. □

COROLLARY 3.4. *The largest cluster to be consumed by another cluster owns at most $n/2$ nodes, the next largest at most $n/3$, etc.*

Thus, we arrive at the following lemma.

LEMMA 3.5. *The total cost of the* DFT *is* $O(n^2 \log n)$ *bits.*

Proof. The cost of traversing a cluster of size k is at most $k^2 \log n$ bits. Hence, the total cost is bounded by $\sum_{i=1}^{n}(\frac{n}{i})^2$ messages. But $\sum_{i=1}^{n}(\frac{n}{i})^2 \leq \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \cdot (\frac{n}{2^i})^2 = \sum_{i=0}^{\lfloor \log n \rfloor}(\frac{n^2}{2^i}) \leq 2n^2$. Hence, the bit complexity of the depth-first traversals is $O(n^2 \cdot \log n)$ bits. □

Adding the bit costs of all three components, we arrive at a total communication complexity of $O(n^2 \log n + n \cdot m)$ bits for the whole algorithm. By the arguments presented in §2.5 we conjecture that this is also the lower bound on the communication complexity of the election problem on arbitrary topology, strongly connected unidirectional networks.

4. The traversal algorithm as a special case of the election algorithm. In this section we show how Traversals 2 and 3 can be derived as a special case of the election algorithm. Imagine the behavior of the election algorithm when it is started by a single node. In this case, the initiator initiates a process which visits all the nodes in the network and terminates. In the first subsection we show that the process can be modified to behave exactly like the process in Traversal-2. In §4.2 we show how Traversal-3 can be derived from the election algorithm.

The process of deriving the traversal algorithms from the election algorithm provides a constructive proof of Lemma 2.8.

4.1. Deriving Traversal-2 from the election algorithm. Assuming that only one node initiates the election algorithm, we make the following observations.

Observation 1: At any given time, at most one cluster is searched, i.e., no messages are exchanged in any of the other clusters.

Observation 2: All the clusters and their selected outgoing links form a simple directed path in which each cluster is a node and each link is a selected outgoing link. This path, called the *clusters active path*, occasionally closes on itself (see Fig. 12). When the path closes either a cycle of clusters is formed, or the last link on the path is an intra-cluster link (in the last cluster on the path, see Fig. 12, cases 2 and 1, respectively).

Observation 3: Consider the cluster's active path when it does not close on itself. Then, the nodes at which the cluster's active path enters other clusters are the first nodes to be explored in each cluster. Therefore, if these nodes were selected as the cluster-heads of their clusters, the active paths of all the clusters would form a single contiguous path at all times.

We now modify the election algorithm according to the above observations, that is, in each cluster the first node to be explored is elected as the cluster-head. Cycle detection (case 2 in Fig. 12) occurs when the token leaves one cluster C_1 (by traversing its selected outgoing link for the first time) and arrives at another, previously explored, cluster C_2. Recalling the third observation, the cluster-head of C_2 is the "oldest" node on the newly formed cycle of clusters. This cluster-head (h) synchronizes the cycle of clusters and is elected to be the cluster-head of the new cluster. Also (from the third observation), the active paths around the cycle form a single contiguous path. If the synchronization phase is modified to leave all the active paths intact, the distinct DFTs may be considered as one DFT. Thus, the newly elected cluster-head h resumes a DFT which already has an active path going through all the clusters around the cycle. As a result, no node in the network is searched more than once during the whole algorithm. The cluster-head resumes the DFT at the node in which the cycle was detected (the LOOP node in Fig. 12, case 2).

The above variation of the election algorithm can be viewed as a traversal process. One node spawns the process which terminates at the initiator after it visits all the nodes in the network one at a time. This traversal process can be further modified to work on a unidirectional network of finite automata, as we show in the next section.

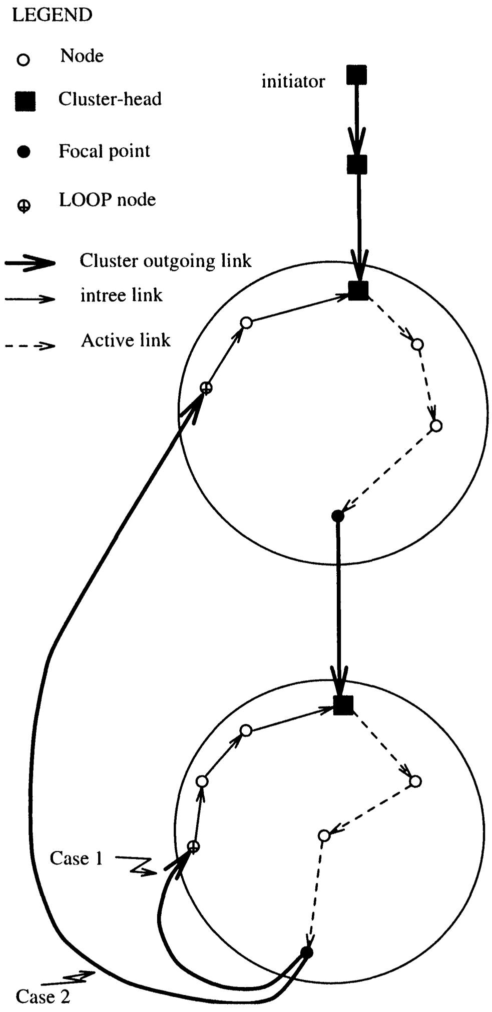


FIG. 12. *Clusters in the traversal algorithm.*

4.2. Deriving Traversal-3 from the election algorithm. Two operations in the preceding traversal algorithm require $O(\log n)$ bits memory in each node. The first is distinguishing an intra-cluster from an inter-cluster link. The second (which occurs during the DFT of the infrastructure) is backtracking from the last node on the active path.

In the first operation, the $O(\log n)$ bits are used to distinguish between the case where a newly traversed link, l, is an intra-cluster link and the case where l closed a cycle of clusters. This operation was accomplished in the preceding algorithm by comparing the ID carried by the token with the cluster-ID of the head node of l. To perform the operation without IDs, we note that, in both cases, l closed a directed cycle of nodes and links (composed of an active path followed by a path in an in-tree, see Fig. 12). In the first case, exactly one cluster-head resides on the cycle, while in the second case, at least two cluster-heads reside on the cycle (see Fig. 12).

We now explain how the token decides whether there is more than one cluster-head on the cycle. The last node on the active path in each cluster is marked as the focal point (it is the focal point of that cluster's DFT, see §2.2.2). The head node of a newly traversed link l is marked LOOP if it is an already explored node. Upon arriving at a LOOP node, the token is sent around the cycle to find out whether there is more than one cluster-head on it. Since there is exactly one LOOP node on such a cycle, the walk around it utilizes a finite number of bits on the token. The token then walks around to the focal point, kills l, and resumes the DFT. If, on the other hand, more than one cluster-head was found, a cycle of clusters was identified. The token then makes another trip around the cycle in order to synchronize its clusters. On the second round, the token first erases all the focal point marks. Second, it erases all the cluster-head marks, aside from the first. Third, the token modifies the in-trees of all clusters, aside from the first one, to be rerooted at the LOOP node (in the same way as was done in the election algorithm). Notice that the active path of the first cluster lies between the first and the second cluster-head. When it arrives at the LOOP node for the second time, the synchronization phase terminates. The LOOP mark is removed, the focal point mark is put on, and the DFT is resumed.

The backtracking in the DFT is performed without using node IDs by using the same solution suggested in §2.3.

The communication cost of the cycle detection and synchronization phases does not change in the modified algorithm. The cost of the cluster outgoing link selection phases is $O(n^2 \log n)$ bits since the DFT searching for outgoing links requires one backtracking for each link of the infrastructure. Thus, the communication complexity of the traversal algorithm is the same as that of the election algorithm.

4.3. Applications.

4.3.1. Applications of the traversal algorithm. In this section we show how the traversal algorithms and their resulting infrastructure can be used to solve other problems. In particular we apply the traversal algorithm to perform broadcasting, route messages, and systematically emulate any bidirectional algorithm on a unidirectional network. Each of the applications can be solved by a traversal; however, after executing the traversal, the application problems may be solved more efficiently using the infrastructure produced by the traversal.

4.3.2. Broadcast with echo. In the broadcast with echo problem one node, the root, has a piece of information which it sends to all the nodes in the network, and the root gets a positive acknowledgment that all the nodes have received the information.

A straightforward solution to the problem will use a traversing process to carry the information on it. The message complexity of this solution is the message complexity of the traversal algorithm, $O(n \cdot m)$.

After one traversal, the next broadcast with echo can be more efficiently performed by traversing only the infrastructure links. Since the infrastructure defines a strongly connected network, any node (not only the root) may start a traversal for this purpose. The complexity of this traversal is $O(n^2)$ since the number of links in the infrastructure is at most $2n - 2$.

After one traversal was performed, a further improvement can be achieved as follows. Every node that wants to start a broadcast with echo first sends the information of the broadcast to the root node along the out-tree marked links, and then the root node starts a broadcast with echo as described below. After receiving the echo, the root node will broadcast an echo on the out-tree links.

The infrastructure can be used for an efficient broadcast with echo from the root as follows. The root sends the broadcast message on all its outgoing links in the infrastructure. Upon receiving the broadcast message for the first time, every other node sends the message to all its outgoing neighbors in the infrastructure. Any other copy of the broadcast message is discarded. This implements the broadcast part of the algorithm. To notify the root that all the nodes in the network have received the broadcast message, every node ν sends an echo as follows. After sending the copy of the broadcast message, ν sends an echo over all of its infrastructure outgoing links except the link marked intree. Node ν sends an echo over the intree marked outgoing link only after an echo has been received on all of its infrastructure incoming links. When the root has received the echo over all of its infrastructure incoming links, the notification has been completed, and a message to this effect is sent on the outtree marked links.

The average message complexity of the resulting broadcast with echo algorithm is $6n$ (averaging out the first broadcast, which is a regular traversal). Sending the broadcast message from the initiating node to the root of the infrastructure costs at most $n - 1$ messages. The broadcast message is then transmitted once over each infrastructure link, which adds at most $2n - 2$ messages to the complexity. The echo message is also sent once over each infrastructure link, hence another $2n - 2$ messages. Then, to pass the echo to the initiating node, another $n - 1$ messages are transmitted on the outtree marked links.

4.3.3. Message sending. The in- and out-trees which result from the traversal algorithm enable us to efficiently send a message from every node to every other node. To pass a message from node ν to node u, node ν sends the message along the intree marked links to the root, which then broadcasts the message on the out-tree. Thus, at most $2n - 2$ messages are sent in the routing mechanism, in order to send a message from any node to any other node.

4.3.4. Transforming bidirectional distributed algorithms to unidirectional networks. The above routing mechanism can be used as a means to emulate any bidirectional distributed algorithm on a strongly connected unidirectional network. Whenever a node has to send a message on an incoming link, it will use the above message passing mechanism. Thus, if a problem has a bit complexity $O(P(n))$ on a bidirectional network, then its complexity on the unidirectional network is upper bounded by $O(n \cdot P(n) + n^2 \cdot \log\ n + n \cdot m)$ (the last two terms entail the construction of the infrastructure).

In case when the bidirectional algorithm is for a network of finite automata, i.e., if nodes are anonymous, then we can send a message backward from node v to its incoming neighbor u by adapting the solution to the "last in the ring" puzzle. However, the adaptation costs $O(m)$ messages (bits) in the synchronous network and $O(m + n \log n)$ messages in the asynchronous network for each message sent backward. The solution is adapted by performing the algorithm on the cycle defined by the unique path from v to the root in the in-tree concatenated with the unique path from the root to u and with the $u \rightarrow v$ link. The messages of the protocols in the proofs of Lemmas 2.11 and 2.12 are marked differently when they go from v to the root and when they travel from the root down to v via the out-tree. Messages going from v to the root behave exactly the same as in the original solutions. Each node participates in the algorithm for a message it receives in the "down" mode over its parent link in the out-tree, and sends the message it would have sent as a response on a ring on all its outgoing links. Any message received by a node other than v, not on the parent out-tree, is ignored, and v responds only to messages it receives from u. This guarantees that messages cycle only on the desired cycle. On all other potential cycles, starting at v the messages are eliminated when they reach a leaf of the out-tree.

5. Concluding remarks.

An interesting observation is that the amount of communication in the unidirectional election algorithm, $O(n \cdot m + n^2 \log\ n)$ bits, is n times the number of messages in the optimal bidirectional algorithm. We would obtain the same cost if we were to simulate the bidirectional algorithm [GHS83], with each acknowledgment charged as n bits, on a unidirectional network. Together with Lemma 2.13, this leads to the conjecture that our election algorithm is as efficient as possible in terms of communication cost.

Acknowledgments. We would like to thank Shimon Even and Peter Winkler for encouraging us to rewrite the results reported here. We would also like to thank an anonymous referee for many helpful corrections.

REFERENCES

[Afe85] Y. Afek, *Distributed Algorithms for Election and Traversal in Unidirectional and Complete Networks*, Ph.D. thesis, University of California, Los Angeles, October 1985.

[AFL83] E. Arjomandi, M. J. Fischer, and N. A. Lynch, *Efficiency of synchronous versus asynchronous distributed systems*, J. Assoc. Comput. Mach., 30 (1983), pp. 449–456.

[AG91] Y. Afek and E. Gafni, *Time and message bounds for election in synchronous and asynchronous complete networks*, SIAM J. Comput., 20 (1991), pp. 376–394.

[Cha79] E. J. Chang, *Decentralized Algorithms in Distributed Systems*, Ph.D. thesis, University of Toronto, October 1979.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh, *An $O(nlogn)$ unidirectional algorithm for extrema finding in a circle*, J. Algorithms, 3 (1982), pp. 245–260.

[ELW90] S. Even, A. Litman, and P. Winkler, *Computing with snakes in directed networks of automata*, In Proc. of the 31st IEEE Annual Symp. on Foundation of Computer Science, Research Triangle Park, NC, pp. 740–745, October 1990.

[Eve79] S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[Fra70] A. S. Fraenkel, *Economic traversal of labyrinths*, Math. Mag., 43 (1970), pp. 125–130.

[GA84] E. Gafni and Y. Afek, *Election and traversal in unidirectional networks*, In Proc. of the 3rd Annual ACM Symp. on Principles of Distributed Computing, August 1984.

[Gal76] R. G. Gallager, *A shortest path routing algorithm with automatic resynch*, Unpublished note, March 1976.

[Gal77] R. G. Gallager, *Finding a leader in a network with $\phi(e) + \phi(n log n)$ messages*, Unpublished note, 1977.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira, *A distributed algorithm for minimum weight spanning trees*, ACM Trans. Program. Lang. Syst., 5 (1983), pp. 66–77.

[GK84] E. Gafni and W. Korfhage, *Distributed election in unidirectional eulerian networks*, In Proc. Twenty-Second Annual Allerton Conference on Communication, Control, and Computing, Allerton, IL, October 1984.

[HT73] J. Hopcroft and R. Tarjan, *Algorithm* 447: *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372–378.

[Hum83] P. A. Humblet, *A distributed algorithm for minimum weight directed spanning trees*. IEEE Trans. Comm., COM-31 (1983), pp. 756–762.

[KKM85] E. Korach, S. Kutten, and S. Moran, *A modular technique for the design of efficient distributed leader finding algorithms*, In Proc. of the ACM Symp. on Principles of Distributed Computing, Minacki, Ontario, Canada, August 1985.

[Kob78] K. Kobayashi, *The firing squad synchronization problem for a class of polyautomata networks*, J. Comput. System Sci., 17 (1978), pp. 300–318.

[Kut84] S. Kutten, Personal communication, 1984.

[Kut88] S. Kutten, *Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed networks*, In Proceedings of the Ninth International Conference on Computer Communication, J. Raviv, ed., pp. 446–452, October 1988.

[Pet82] G. L. Peterson, *An $\phi(n log n)$ unidirectional algorithm for the circular extrema problem*, ACM Trans. Program. Lang. Syst., 4 (1982), pp. 758–762.

[Seg83] A. Segall, *Distributed network protocols*, IEEE Trans. Inform. Theory, IT-29 (1983), pp. 23–35.

[Tar72] R. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1179–1192, December 1994

SIMPLE AND FAST ALGORITHMS FOR LINEAR AND INTEGER PROGRAMS WITH TWO VARIABLES PER INEQUALITY*

DORIT S. HOCHBAUM† AND JOSEPH (SEFFI) NAOR‡

Abstract. The authors present an $O(mn^2 \log m)$ algorithm for solving feasibility in linear programs with up to two variables per inequality which is derived directly from the Fourier–Motzkin elimination method. (The number of variables and inequalities are denoted by n and m, respectively.) The running time of the algorithm dominates that of the best known algorithm for the problem, and is far simpler. Integer programming on monotone inequalities, i.e., inequalities where the coefficients are of opposite sign, is then considered. This problem includes as a special case the simultaneous approximation of a rational vector with specified accuracy, which is known to be NP-complete. However, it is shown that both a feasible solution and an optimal solution with respect to an arbitrary objective function can be computed in pseudo-polynomial time.

Key words. linear programming, integer programming

AMS subject classifications. 05C85, 68Q25, 90C05, 90C10, 90C27

1. Introduction.

In this paper we examine linear and integer programming problems with two variables per inequality. The problem of computing a feasible solution in the linear (or fractional) case has been investigated extensively. Shostak [18] suggested that a linear program with two variables per inequality can be represented as a graph: since each inequality contains two variables, one can represent the linear program by a graph which has a vertex for each variable, and an additional vertex x_0. Any inequality involving two variables is represented as an edge between the respective pair of vertices. As for inequalities involving only one variable (upper and lower bounds on variables), these are represented as edges to and from vertex x_0. We denote the number of variables by n, and the number of inequalities by m. (W.l.o.g. we can assume that $m \geq n$.) The graph consists therefore of $n + 1$ vertices and m edges, and there may be multiple edges between any pair of vertices.

Shostak [18] proved that feasibility can be tested by following paths and cycles in this graph, and thus laid the foundation for all subsequently considered algorithms for the problem. This feasibility test was refined to a polynomial algorithm by Aspvall and Shiloach [1], and later still to an $O(mn^3 \log m)$ strongly polynomial algorithm by Megiddo [14]. Recently, Cohen and Megiddo [4] obtained new algorithms for this problem: (i) they presented a new $O(mn^2(\log m + \log^2 n))$ time algorithm; (ii) they also gave a randomized algorithm for finding a feasible solution in the special case of monotone inequalities (to be defined later) with an expected running time of $O(n^3 \log n + mn \log^3 m \log n + mn \log^5 n)$. (This randomized algorithm was later generalized to hold for the non-monotone case as well in [2]; however, it follows from [5] that the non-monotone case can be reduced to the monotone case at no extra cost.) The main feature common to all of these algorithms is determining upper and lower bounds for each variable by following paths and cycles in the graph.

The first result we present is an $O(mn^2 \log m)$ algorithm for the feasibility problem. This algorithm is faster (although only for $m < n^{o(\log n)}$), and moreover it is *simpler* than all other known algorithms. The backbone of our algorithm is the Fourier–Motzkin elimination method

*Received by the editors July 1, 1993; accepted for publication July 9, 1993. A preliminary version of this paper appeared in the Proceedings of the 2nd "Integer Programming and Combinatorial Optimization" Conference (IPCO), CMU, 1992, pp. 44-59.

†Department of Industrial Engineering and Operations Research, University of California, Berkeley, California 94720 (dorit@hochbaum.berkeley.edu). This research was supported in part by Office of Naval Research grant ONR N00014-91-J-1241.

‡Department of Computer Science, Technion, Haifa 3200, Israel (naor@cs.technion.ac.il). Most of this work was done while the author was at the Computer Science Department, Stanford University and supported by contract ONR N00014-88-K-0166. The author was also supported in part by contract ONR N00014-91-J-1241.

(Introduced by Fourier (1827), and discovered later by Dines (1918–1919) and Motzkin (1936); see [17] for details.) In general, this algorithm does not run in polynomial time because it may generate an exponential number of inequalities in the process of eliminating variables. However, we show how to implement this algorithm *efficiently* for linear programs where each inequality may contain at most two variables. First, at each elimination step, the number of inequalities on every edge adjacent to the variable currently to be eliminated is reduced to two. This serves to control the exponential growth of the number of inequalities. In addition, we maintain the inequalities corresponding to two variables as upper and lower envelopes, where the envelopes (which are piecewise linear functions) are characterized by their breakpoints. This representation allows us to dispose of redundant inequalities in each elimination step quickly by examining all breakpoints associated with the variable currently to be eliminated.

The analogue of the Fourier–Motzkin procedure in computational logic is *resolution.* Using resolution, one obtains a satisfying assignment to a set of clauses (in, say, propositional logic) by eliminating the variables one by one. It is known that resolution can be efficiently implemented for the case of 2-SAT clauses, i.e., the satisfiability problem, where each clause may contain at most two literals. This follows since every elimination step generates 2-SAT clauses, and the total number of 2-SAT clauses is always bounded by a polynomial. Our algorithm may be viewed as an efficient implementation of resolution for the case of linear constraints with two variables per inequality.

A linear program with two variables per inequality is called *monotone* if each inequality is of the form $ax_i - bx_j \leq c$, where both a and b are positive. We will consider integer programming problems on monotone inequalities. We note that the aforementioned reduction from the non-monotone case to the monotone case does not preserve integrality.

Lagarias [11] has shown that the problem of deciding whether a given rational vector α has a simultaneous approximation of specified accuracy with respect to the maximum norm, with denominator Q in a given interval $1 \leq Q \leq N$, is NP-complete. The problem of deciding the feasibility of a monotone system in integers is a generalized form of this question and hence NP-complete as well (it is obviously in NP).

The set of feasible solutions of a monotone system can be shown to form a *distributive lattice* where the *join* and *meet* operations are defined to be maximum component-wise and minimum component-wise, respectively. This has been observed before by Veinott [20]. We present an algorithm that computes the solution vectors corresponding to the top and bottom of the lattice. The lattice structure is crucial for the algorithm, and the manner in which the search for a feasible solution is conducted guarantees that if one exists, then we are going to find the solution which is at the top (or bottom). The running time of this algorithm is a polynomial which depends on the sum of the number of integer valued points in each one-dimensional projection of the feasible polytope in the fractional case. Hence, this algorithm is pseudo-polynomial in the case when the variables in the integer program are bounded. Also, in this case the problem is weakly NP-complete.

It is interesting to note that the strongly polynomial feasibility algorithm for linear inequalities with two variables per inequality does not extend to a strongly polynomial *optimization* algorithm over such inequalities. (It is only known that when the objective function consists of d variables, then there is a strongly polynomial algorithm when d is fixed, i.e., it is exponential in d.)

In contrast, for the integer case, we present a pseudo-polynomial algorithm for the optimization problem over a monotone system with an arbitrarily long objective function (that is, with up to n variables in the objective). We note that the optimization problem over a non-monotone system is NP-complete in the strong sense, since the vertex cover problem is a special case of it. The algorithm hinges on the following two observations:

- The elements of a distributive lattice can be represented as closed subsets of a directed graph (of pseudo-polynomial size) which is derived from the lattice.
- A linear objective function defines a modular function on the lattice which in turn implies that the lattice element of optimal cost corresponds to the closed subset of optimal cost (when costs are properly defined).

The complexity of computing the closed subset of optimal cost is bounded by a polynomial in the size of the graph, i.e., it can be computed in pseudo-polynomial time. Even though the directed graph that represents the lattice is of pseudo-polynomial size, it has a succinct description, i.e., it can be encoded in polynomial space. This provides a compact encoding of the complete feasible solution set of a monotone system of inequalities.

Finally, we present an application of our Fourier–Motzkin algorithm to identifying *fat* polytopes. Fat polytopes are those containing a sphere which circumscribes a unit hypercube, and hence must contain an integer point. For polytopes derived from inequalities with two variables per inequality, the procedure for identifying fat polytopes runs in strongly polynomial time, and thereby can be viewed as an efficient heuristic for finding a feasible integer solution. A strongly polynomial algorithm for a related problem of finding the largest sphere contained in a polytope is presented as well.

2. Efficient implementation of the Fourier–Motzkin algorithm. In this section we show how the Fourier–Motzkin elimination method for finding a feasible solution of a linear program can be implemented efficiently when the number of variables in each inequality is at most two. We begin by an informal description of the method for a general linear program. (The reader is referred to [17, pp. 155–156] for more details.)

Let the variables of the linear program be $x_1, \ldots, x_n$ and let the set of inequalities be denoted by E. The variables are eliminated one by one. At step i, the linear program will only contain variables $x_i, \ldots, x_n$; the set of inequalities at step i is denoted by E_i, where initially $E_1 = E$. To eliminate variable x_i, all the inequalities in which x_i participates are partitioned into two sets, L and H. The set L contains all the inequalities which are of the form $x_i \geq l$, and the set H contains all the inequalities which are of the form $x_i \leq h$, where l and h are linear functions. To obtain the set E_{i+1}, for all $l \in L$ and $h \in H$, a new inequality $l \leq h$ is added to E_i, and all the inequalities in L and H are eliminated from it. The number of new inequalities produced is $|H| \cdot |L|$. The next theorem is immediate.

THEOREM 2.1. *The linear program E_{i+1} has a feasible solution if and only if the linear program E_i has a feasible solution.*

Hence, a feasible solution can be computed recursively for E_{i+1} and then extended to E_i. The main drawback of this method is that the running time is not necessarily polynomial, i.e., in general the number of inequalities may grow exponentially.

The discussion henceforth is restricted to inequalities that contain at most two variables per inequality. It is interesting to note that Nelson [15] proved that when implementing the Fourier–Motzkin method in this case, the total number of inequalities is bounded by $m \cdot n^{\log n}$.

As mentioned in the introduction, an equivalent representation of the linear program is by the graph $G = (V, E)$. The vertex set V contains vertices $x_0, x_1, \ldots, x_n$; an edge between vertex x_i and x_j (for $1 \leq i, j \leq n$) represents the set of inequalities in which x_i and x_j participate. The vertex x_0 is needed to represent inequalities that contain precisely one variable, i.e., an edge from x_i to x_0 denotes an inequality of the form $x_i \leq a$ or $x_i \geq a$ for some constant a.

The main feature that allows for the efficient implementation of the Fourier–Motzkin algorithm is the following. The set of inequalities that correspond to an edge between x_i and x_j is represented in the (x_i, x_j) plane as two envelopes, an upper envelope and a lower envelope. The feasible region of x_i and x_j is in between the two envelopes and it is not hard to see that each envelope is a *piecewise linear* function that can be represented by its breakpoints (see Fig. 1(a)).

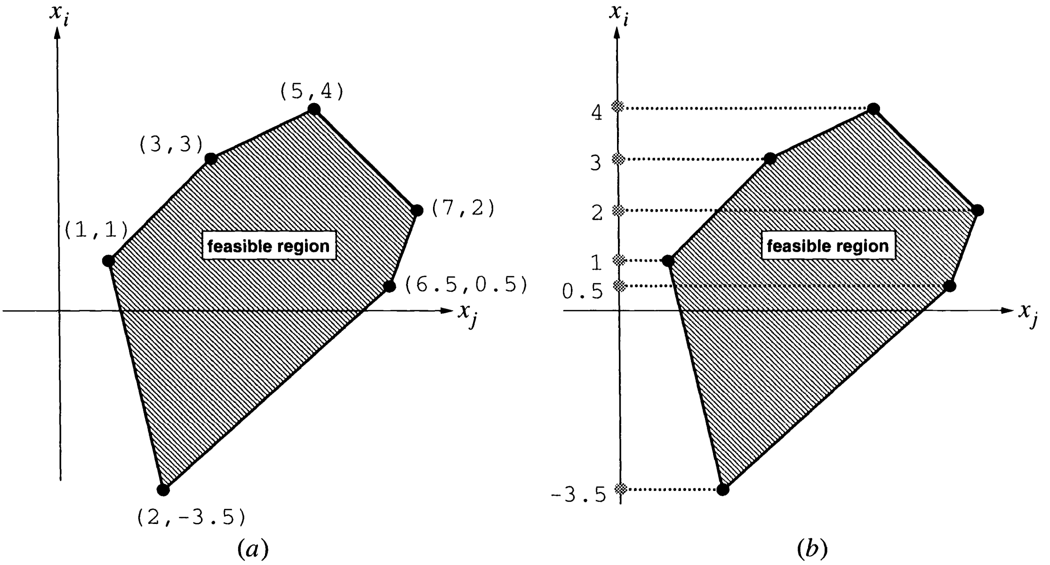


FIG. 1. (a) *The feasible region defined by the inequalities containing x_i and x_j is a piecewise linear function defined by its breakpoints.* (b) *The set $B_j = \{-3.5, 0.5, 1, 2, 3, 4\}$ is the set of breakpoints projected on the x_i axis.*

The following procedure of Aspvall and Shiloach [1] plays a crucial role in our algorithm. This procedure was used by [14] and [4] as well. Let $x_i^{\min}$ and $x_i^{\max}$ denote the respective minimum and maximum feasible values of x_i. That is, any value assigned to x_i from the range $[x_i^{\min}, x_i^{\max}]$ can be complemented to a feasible solution. If the feasible range of some of the variables is unbounded, then there exist numbers bounded by a polynomial in the binary representation of the data [17], such that if $x_i^{\min}$ and $x_i^{\max}$ are set to them, the existence of a feasible solution is assured.

PROCEDURE 2.1 [1]. *Given a variable x_i and a value λ, it can be decided in $O(mn)$ operations whether (i) $\lambda < x_i^{\min}$, (ii) $\lambda > x_i^{\max}$, or (iii) $x_i^{\min} \leq \lambda \leq x_i^{\max}$.*

The main idea underlying Procedure 2.1 is propagating the implications of the equality $x_i = \lambda$ in a manner very similar to the Bellman–Ford algorithm for computing all shortest paths from a single source. We remark that even if the linear program in hand is infeasible, the procedure may still provide one of the above three answers. In this case, infeasibility will be detected by our algorithm at a later stage. (Note for example the case in which the linear system consists of two independent subsystems, one feasible and one infeasible.)

We are now ready to provide a high-level view of the algorithm. The main idea is that the number of inequalities in which x_i (the variable to be eliminated) participates can be significantly reduced using Procedure 2.1. It should be mentioned that a similar idea was used by Megiddo [14] to obtain upper and lower bounds on the feasible values of variables. The following is performed at step i of the Fourier–Motzkin algorithm. Let G_i denote the graph corresponding to the linear program E_i.

1. Let the neighbors of x_i in the graph G_i be $x_{i_1}, \ldots, x_{i_d}$.
   Let B_j $(1 \leq j \leq d)$ denote the set of breakpoints of the edge (x_i, x_{i_j}) projected on the x_i coordinate (see Fig. 1(b)).

2. Merge the d sorted sequences B_i into a sorted sequence B. (Let the sorted sequence be $b_1, \ldots, b_k$.)
3. Perform a binary search on the sequence B. The aim of the search is to obtain either
   (a) a breakpoint $b_l \in B$ such that $x_i^{\min} \leq b_l \leq x_i^{\max}$, or
   (b) an interval $[b_l, b_{l+1}]$ $(1 \leq l < k)$ such that $b_l < x_i^{\min}$ and $x_i^{\max} < b_{l+1}$.
4. In step 3a, variable x_i is assigned the value b_l and "contracted" with vertex x_0 in graph G_i.
   In step 3b, the number of inequalities on each edge adjacent to x_i is reduced to at most two (see Fig. 2). Now, the generic Fourier–Motzkin elimination step is applied to variable x_i.

Let us further elaborate on how the algorithm is implemented and analyze its complexity. The following invariant is maintained throughout the algorithm; we defer its proof to the end of the discussion. It is obviously true initially.

INVARIANT 2.1. *The number of breakpoints on an edge is at most $O(m)$.*

By the invariant, the cardinality of the set B is at most $O(mn)$. The binary search at step 3 is performed by successive calls to Procedure 2.1. At each call, either a breakpoint which is feasible for x_i is discovered, or the number of breakpoints to be examined is reduced by half. Hence, the complexity of sorting the set B and performing the binary search is at most $O(mn \log m)$. We should remark that in the course of the elimination process, to bound the running time of Procedure 2.1 by $O(mn)$, we run it on the original graph G and not on the current graph G_i. However, G is updated as follows. For each eliminated variable (say x) that was assigned a value (say a) at step 3a, two inequalities are added to graph G: $x \leq a$ and $x \geq a$. If x is already connected to x_0, then the respective bounds are updated according to the most restrictive bound, or an inconsistency is detected and the algorithm terminates with a discovery that the system is infeasible.

In step 3a, the linear program E_{i+1} is obtained from E_i by assigning the value b_l to the variable x_i. Otherwise, in step 3b, the generic Fourier–Motzkin elimination step is applied. Notice that the number of inequalities on each edge adjacent to x_i is reduced to at most two (see Fig. 2). (We assume that the intersections of the upper and lower envelopes (up to two) are also counted among the original breakpoints.) In addition, two more inequalities, $b_l \leq x_i$ and $x_i \leq b_{l+1}$, are added to the linear program E_i.

Let x_{i_p} and x_{i_q} be any two variables that are adjacent to x_i. The edge (x_i, x_{i_q}) and the edge (x_i, x_{i_q}) may each contain at most two inequalities; hence, at step 4, the Fourier–Motzkin elimination step adds up to four new inequalities between the variables x_{i_p} and x_{i_q}. These four inequalities are added to the set of inequalities that already exist between them. The running time of adding a new inequality to an already existing envelope is $O(\log m)$ time. This follows since the existing set of inequalities includes at most $O(m)$ inequalities (Invariant 2.1), which is represented as an (upper and lower) envelope, i.e., as a sorted sequence of breakpoints. Adding a new inequality amounts to identifying where to insert the newly created breakpoint in the existing sequence, which can be done using a binary search. We note that it may be the case that, as a result of adding a new breakpoint, many other breakpoints can disappear. Since there are at most $\binom{n}{2}$ pairs of neighbors, the complexity of this step is at most $O(n^2 \log m)$.

To prove Invariant 2.1, notice that for each variable that is eliminated, the number of breakpoints added to an edge is a constant, and hence the invariant is maintained. In fact, the number of breakpoints on an edge will never exceed $m + 4n$ throughout the execution of the algorithm.

At the end of the elimination step, we are left with two variables, x_0 and x_n. We now backtrack and assign values to the inequalities as follows. Choose any feasible value in the feasible range for x_n. Now choose a feasible value for x_{n-1}, that satisfies the inequalities w.r.t.

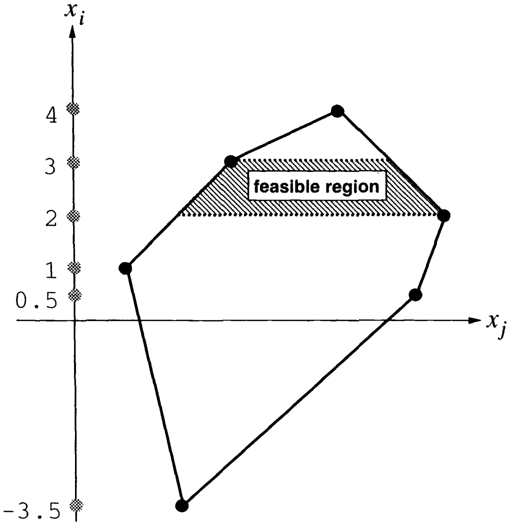


FIG. 2. *Step* 3b *of the algorithm: for example,* $b_l = 2$ *and* $b_{l+1} = 3$. *Consequently, the number of inequalities involving* x_i *and* x_j *is reduced to two. In addition, there are two more inequalities:* $2 \leq x_i$ *and* $x_i \leq 3$.

x_n and x_0. Continue inductively by determining a value for x_i, based on the inequalities of x_i and x_j for $j > i$, and the range determined by the upper and lower bound inequalities with x_0. Since there are now at most two inequalities on each edge, the running time for the entire backtracking process is $O(n^2)$.

The correctness of the next theorem follows from the above discussion.

THEOREM 2.2. *The complexity of eliminating a variable in the algorithm is* $O(mn \log m)$. *Hence, the complexity of the entire algorithm is* $O(mn^2 \log m)$.

3. Integer programming on monotone inequalities. A linear program with two variables per inequality is called *monotone* if for every inequality, the coefficients of the two variables have opposite signs. We begin by studying the properties of the set of feasible vectors in the case of monotone inequalities. This set can be looked upon as a partial order under the following definition of dominance. Given two feasible vectors, L_1 and L_2, we say that $L_1 \succeq L_2$ if for all components i, $L_1(i) \geq L_2(i)$. Let $\mathcal{L}$ denote the set of all feasible vectors of a monotone system. We prove that $\mathcal{L}$ has the nice property that it forms a *distributive lattice*. This property will turn out to be very useful for finding a feasible solution and optimizing with respect to an objective function. It was previously observed by Veinott [20]. A distributive lattice is a partial order in which

1. Each pair of elements has a greatest lower bound, or *meet*, denoted by $a \wedge b$, so that $a \wedge b \preceq a$, $a \wedge b \preceq b$, and there is no element c such that $c \preceq a$, $c \preceq b$ and $a \wedge b \prec c$.
2. Each pair of elements has a least upper bound, or *join*, denoted by $a \vee b$, so that $a \preceq a \vee b$, $b \preceq a \vee b$, and there is no element c such that $a \preceq c$, $b \preceq c$ and $c \prec a \vee b$.
3. The *distributive* laws hold, namely $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ and $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$.

To prove that $\mathcal{L}$ is, in fact, a distributive lattice, we define appropriately the *meet* and *join* operations. The meet of two vectors L_1 and L_2 is defined to be the vector where each component is the minimum of the two corresponding components in L_1 and L_2. The join of two vectors is defined similarly where minimum is replaced by maximum. (See Fig. 3 for an example.)

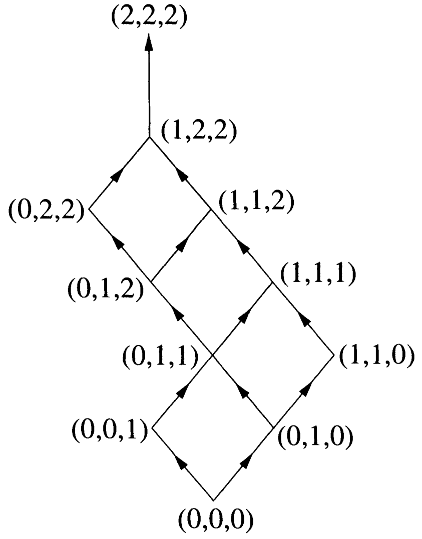
THEOREM 3.1 [20]. *The partial order* $(\mathcal{L}, \preceq)$ *of feasible vectors forms a distributive lattice under the above definitions of meet and join.*

Proof. We first establish that $(\mathcal{L}, \preceq)$ is a lattice. To do that, we prove one case, other cases follow similarly. Let $L_1 = (u_1, \dots, u_n)$, $L_2 = (v_1, \dots, v_n)$, and let $L = L_1 \vee L_2 = (w_1, \dots, w_n)$. We show that L is also a feasible solution vector. For a particular inequality $ax_i - bx_j \le c$, we know that

$$au_i - bu_j \le c; \quad av_i - bv_j \le c.$$

If, for example, $w_i = u_i$ and $w_j = v_j$, then since b is positive, $bu_j \le bv_j$, and the inequality holds for solution vector L as well.

Let a, b, and c be any integers. Then, $\min(a, \max(b, c)) = \max(\min(a, b), \min(a, c))$ and $\max(a, \min(b, c)) = \min(\max(a, b), \max(a, c))$. Hence, the distributive laws hold for the lattice $\mathcal{L}$. □



$$\begin{aligned} y - x &\le 2 \\ x - y &\le 0 \\ 2z - y &\le 3 \\ 2y - z &\le 2 \end{aligned}$$

$$0 \le x,\ y,\ z \le 2$$

FIG. 3. *The sublattice of integral solutions of a monotone system. Each solution vector is of the form* (x, y, z).

Notice that the lattice property holds in both the fractional and the integer case, and in fact the set of integer feasible solutions is a *sublattice* of the lattice of feasible solutions. From now on $\mathcal{L}$ will denote the lattice in the integer case and we restrict the discussion to this lattice. It is easy to see that the lattice properties imply that a lattice has a unique minimum and maximum, denoted by B (bottom) and T (top), respectively.

The problem of checking whether an integer monotone system has a feasible solution was shown to be NP-complete by Lagarias [11]. This was shown by proving that the following problem, *good simultaneous approximation*, is NP-complete. An instance of this problem consists of a vector of rationals, $\alpha = (a_1/b_1, \dots, a_n/b_n)$, and positive integers N, s_1, and s_2. The question is whether there exists an integer Q, $1 \le Q \le N$, such that

$$\max_{1 \le i \le n} \{Qa_i/b_i\} \le \frac{s_1}{s_2},$$

where $\{\beta\}$ denotes the distance of β to the closest integer. This problem can be expreseed as an instance of finding an integer feasible solution $(x_1, \dots, x_n, Q)$ for the following monotone system:

$$-s_1 \cdot B \leq s_2 \left(\frac{a_i \cdot B}{b_i} \cdot Q - B \cdot x_i \right) \leq s_1 \cdot B, \qquad 1 \leq i \leq n,$$

$$\text{where} \quad B = b_1 \cdot b_2 \cdots b_n \qquad \text{and} \qquad 1 \leq Q \leq N.$$

However, we will show in §§3.1 and 3.2 that for the case of bounded variables, both the feasibility problem and the optimality problem with respect to an arbitrary objective function can be solved in pseudo-polynomial time over a monotone system of inequalities. Consequently, integer programming over monotone inequalities with bounded variables is only weakly NP-complete.

3.1. Integer feasibility over monotone inequalities. In this section we show how a feasible integer solution can be found. More specifically, our aim is to compute the feasible solution which corresponds to the top of the lattice, i.e., the feasible vector whose components are maximal. The same procedure with a slight modification can be applied to find the solution corresponding to the bottom of the lattice.

During the course of this procedure a current solution vector $L = (x_1, \ldots, x_n)$ (which is infeasible) is maintained with the invariant that $L \succeq T$. The initial value of L is the top of the fractional lattice where each component is rounded downward to the nearest integer.

It should be noted that in the monotone case, the top (or bottom) of the lattice in the fractional case can be computed via a simple modification of the algorithm defined in §2. In step 3a of the algorithm, instead of assigning a value to the variable that is eliminated, two consecutive breakpoints are computed with the following property. One breakpoint belongs to the feasible region, and the other breakpoint (the larger one) does not belong to it. The algorithm continues similarly to step 3b. In the backtracking process, the highest feasible value is chosen for each variable.

The generic step in the algorithm is as follows. Traverse all the inequalities in the linear program in some arbitrary order. If an inequality $ax_i - bx_j \leq c$ is invalid, then it is *validated* by updating the value of x_i as follows:

$$x_i \leftarrow \left\lfloor \frac{c + bx_j}{a} \right\rfloor.$$

The algorithm terminates if either all inequalities traversed in a single step are valid, or $L \preceq B$.

THEOREM 3.2. *The algorithm tests in time $O(mn^2 \log m + m \cdot \sum_{i=1}^{n}(x_i^{\max} - x_i^{\min} + 1))$ whether a monotone system of inequalities has a feasible solution.*

Proof. Assume that the given system of inequalities has a feasible solution. We show that the invariant that $L \succeq T$ is maintained throughout the algorithm. Given an invalid inequality $ax_i - bx_j \leq c$, where $x_i = u_i$ and $x_j = u_j$, it is validated by decreasing x_i. Assume that the values of x_i and x_j in T are $\hat{x}_i$ and $\hat{x}_j$, respectively. By the invariant, $u_j \geq \hat{x}_j$. Hence,

$$\hat{x}_i \leq \left\lfloor \frac{c + b\hat{x}_j}{a} \right\rfloor \leq \left\lfloor \frac{c + bu_j}{a} \right\rfloor = u_i$$

and the invariant is maintained.

It follows from the invariant that we will never need to backtrack, and since the value of a variable is always decreased by at least one unit whenever an inequality is validated, the running time of the algorithm is bounded by $O(mn^2 \log m + m \cdot \sum_{i=1}^{n}(x_i^{\max} - x_i^{\min} + 1))$. This running time is pseudo-polynomial since the feasible range of the variables is bounded.

If the monotone system does not have an integer feasible solution, this will be detected when $L \preceq B$, since $T \succeq B$ must hold. □

Recently, T. Feder (private communication) observed that an integer feasible solution can be computed in pseudo-polynomial time in the non-monotone case when the variables are bounded.

3.2. Integer optimization over monotone inequalities. In this section we consider the following integer optimization problem:

$$\min \quad \sum_{i=1}^{n} w_i \cdot x_i$$

$$\text{subject to} \quad a_k x_i - b_k x_j \leq c_k, \quad k = 1, \ldots, m,$$

$$1 \leq i, j \leq n \quad \text{and} \quad a_k,\ b_k \geq 0.$$

We show that the optimal solution can be computed in pseudo-polynomial time where the polynomial depends on $\sum_{i=1}^{n}(x_i^{\max} - x_i^{\min})$. Recall that the optimization problem over a non-monotone system is NP-complete in the strong sense since vertex cover is a special case of it.

DEFINITION 3.3. *Let f be a function defined on a lattice $\mathcal{L}$, and let $a, b \in \mathcal{L}$. The function f is called modular if $f(a) + f(b) = f(a \vee b) + f(a \wedge b)$.*

It is straightforward to verify that any linear objective function defined on an integer monotone system is modular.

DEFINITION 3.4. *For a directed graph G, a subset S is said to be closed if for every $s \in S$, all its predecessors, i.e., all vertices s' for which there exists a directed path from s' to s, belong to the subset S.*

We first review our scheme for minimizing with respect to an objective function. The following theorem in lattice theory (see [6, p. 72, Thm. 9] and [7, Thm. 2.2.1]) is relevant to our result.

THEOREM 3.5. *Given a distributive lattice $\mathcal{L}$, a partial order can always be associated with it, such that a one-to-one correspondence can be established between its closed subsets and the elements of $\mathcal{L}$.*

The proof of this theorem is constructive and it implies an algorithm for constructing the partial order. In general, there may be more than one partial order that has the above property; we denote by $I(\mathcal{L})$ the partial order obtained by following the proof of Theorem 3.5 and call it the *generic partial order*.

Suppose now that a modular function f is defined on the lattice $\mathcal{L}$. It can be shown that in this case, the elements of the partial order can be assigned costs in such a way that the lattice element of optimal cost would correspond to the closed subset of $I(\mathcal{L})$ of optimal cost. Computing a closed subset of optimal cost in a partial order is a well-known problem and its complexity is bounded by a polynomial in the size of $I(\mathcal{L})$ [16]. (The size of $I(\mathcal{L})$ is pseudo-polynomial in the case of our lattice.)

The disadvantage of computing with the generic partial order $I(\mathcal{L})$ is that its structure is rather complicated, and it seems that it cannot be described compactly, i.e., in polynomial space (as opposed to pseudo-polynomial space). Instead, we present a directed graph, denoted by $G(\mathcal{L})$, that also has the property that a one-to-one correspondence exists between its closed subsets and the elements of $\mathcal{L}$. The advantage of this graph is that it can be encoded in polynomial space via an algorithm which has a short (polynomial) description.

The rest of the section is organized as follows. In §3.2.1 we define the directed graph $G(\mathcal{L})$ and prove that it has the desired properties. In §3.2.2 we show how to compute a closed subset of minimum cost in $G(\mathcal{L})$. For the sake of completeness, we discuss in §3.2.3 how to obtain the graph $G(\mathcal{L})$ from the generic partial order $I(\mathcal{L})$.

It should be noted that similar methods were used by Gusfield and Irving [7] to compute efficiently an *egalitarian* solution for the Stable Marriage problem. See also [9], [19] for an application of these methods.

3.2.1. Constructing the directed graph. In this section we define a directed graph $G(\mathcal{L})$ such that a one-to-one correspondence can be defined between its closed subsets and the elements of $\mathcal{L}$. (See Fig. 4 for an example.)

Let the set V_i be defined as the set of integers that are contained between the largest and smallest integer feasible values of variable x_i. The vertex set of $G(\mathcal{L})$ is $V_1 \cup \cdots \cup V_n$, i.e., a vertex is created for each $v \in V_i$, where $1 \leq i \leq n$. In addition, there is a special vertex denoted by s. The edge set of $G(\mathcal{L})$ is defined as follows.

- For each variable x_i, a directed chain is defined on the set V_i in sorted order. That is, for each pair of vertices representing two consecutive values, v and $v + 1$, there is an arc $(v, v + 1)$. Such a chain is called an x_i-chain.
- For each inequality $ax_j - bx_i \leq c$, the following "ladder" is defined between the x_i-chain and the x_j-chain. For all $v \in V_j$, there is an arc from the value corresponding to $\lceil \frac{av-c}{b} \rceil$ in the x_i-chain, to the vertex corresponding to v in the x_j-chain. Intuitively, the arcs can be thought of as constraints, i.e., if the value of variable x_j is v, then the value of variable x_i must be at least $\lceil \frac{av-c}{b} \rceil$ in any feasible solution
- For each x_i-chain, there is a bidirectional edge connecting the vertex corresponding to the smallest value in V_i to the vertex s. The purpose of these edges is to ensure that any closed set contains at least one vertex in each x_i-chain.

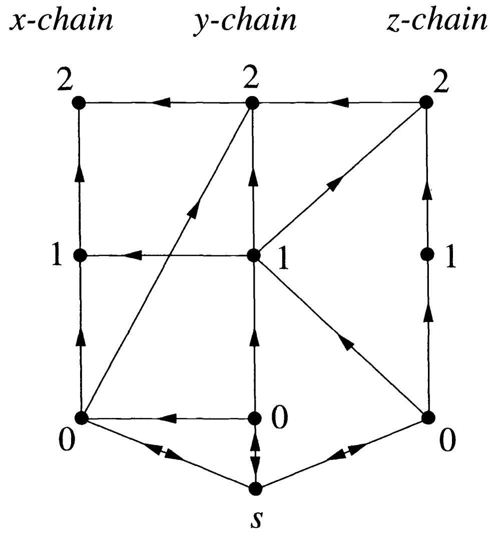


FIG. 4. *The directed graph $G(\mathcal{L})$ corresponding to the set of inequalities of Fig. 3. For example, the arc connecting the "0" value in the x-chain to the "2" value in the y-chain is implied by the inequality $y - x \leq 2$.*

The next theorem states that this construction is valid.

THEOREM 3.6. *There is a one-to-one correspondence between the closed subsets of $G(\mathcal{L})$ that contain vertex s and the elements of $\mathcal{L}$.*

Proof. We first prove that every feasible solution vector L defines a closed subset in $G(\mathcal{L})$. Let $L = (u_1, \ldots, u_n)$. The corresponding subset S_L in $G(\mathcal{L})$ is defined by taking for all i, all the vertices corresponding to integers that are smaller or equal than u_i in V_i, and $s \in S_L$. Assume now that S_L is not a closed subset. Then there exist two vertices, $v \in V_i$ and $w \in V_j$, such that

- $w \in S_L$ and $v \notin S_L$, and,
- there is an arc from v to w in $G(\mathcal{L})$.

This arc can only be generated by the inequality $ax_j - bx_i \leq c$. (We now abuse terminology by treating v and w as both vertices and integers.) Thus, by the construction of $G(\mathcal{L})$, $v = \lceil \frac{aw-c}{b} \rceil$; since $u_i \geq \lceil \frac{au_j-c}{b} \rceil$ and $w \leq u_j$, we get that $v \leq u_i$ and hence $v \in S_L$, contradicting our assumption.

We now prove the other direction. Let S_L be a closed subset in $G(\mathcal{L})$. The solution vector $L = (u_1, \ldots, u_n)$ corresponding to S_L is defined by taking u_i to be the largest integer belonging to S_L in V_i. Since $s \in S_L$, then every x_i-chain in the graph $G(\mathcal{L})$ has at least one representative in the closed subset S_L. Suppose L is infeasible, i.e., there exists an inequality $ax_j - bx_i \leq c$ such that $au_j - bu_i > c$. By the construction of $G(\mathcal{L})$, there is an arc from the vertex in V_i corresponding to $\lceil \frac{au_j-c}{b} \rceil$ to the vertex in V_j corresponding to u_j. Since S_L is a closed subset, the vertex corresponding to $\lceil \frac{au_j-c}{b} \rceil$ must belong to S_L, and hence, $u_i \geq \lceil \frac{au_j-c}{b} \rceil$, which contradicts the invalidity of the inequality. □

REMARK 3.1. *Notice that $G(\mathcal{L})$ has a succinct description and can be encoded in polynomial space. This implies that the complete set of solutions of a monotone system can be completely encoded in polynomial space.*

3.2.2. Minimizing with respect to a modular function. In this section we discuss how to compute a lattice element minimizing an objective function $\sum_{i=1}^{n} w_i x_i$. As mentioned earlier, this objective function also defines a modular function on the lattice. We first show how to assign costs to the vertices of $G(\mathcal{L})$ such that a closed subset of minimum cost corresponds to a lattice element of minimum cost. Then we briefly review Picard's algorithm [16] for finding a closed subset of minimum cost in a directed graph.

The cost of every vertex in $G(\mathcal{L})$ is determined as follows. Let the smallest value in V_i be b_i. The cost of the vertex corresponding to b_i is $w_i \cdot b_i$, and the cost of the other vertices in V_i is w_i. It is not hard to see that finding the optimal solution with respect to an objective function is equivalent to finding the closed subset of minimum cost in $G(\mathcal{L})$, where the cost of a closed set is defined to be the sum of the costs of its members. The problem of computing the minimum cost closed set can be reduced to computing the minimum cut in the following graph (of pseudo-polynomial size), denoted by G, which is derived from $G(\mathcal{L})$. (Computing the minimum cut in G can be done by finding the maximum flow from the source to the sink.)

- Connect all positive cost vertices to a source and all negative cost vertices to a sink.
- The capacity assigned to edges adjacent to the source or sink is equal to the absolute value of the cost of the vertices to which they are adjacent.
- All other edges in $G(\mathcal{L})$ have infinite capacity in G.

By our construction, the minimum cut must either contain edges adjacent to the source or to the sink. (Other edges have infinite capacity.) The *sink-set* of a cut is defined to be the set of vertices that can be reached from the source only via the cut. Picard [16] proved that the sink-set defined by the minimum cut in G corresponds to a closed subset of minimum cost in $G(\mathcal{L})$. To see that, let N be the sum of the capacities of the edges adjacent to the source in G. It is not hard to see that the cost of the vertices in the sink-set of any finite cut is equal to $-N$ plus the capacity of the cut. Hence, a minimum cut defines a closed subset of minimum cost.

Since a minimum cut can be identified in a graph $G = (V, E)$ in $O(|E||V| \log |V|)$, e.g., [8], and in our graph $|V| = O(\sum_{i=1}^{n} |V_i|)$ and $|E| \leq O(m \sum_{i=1}^{n} |V_i|)$, we have the following theorem.

THEOREM 3.7. *The integer optimal solution of a monotone system of inequalities with respect to an arbitrary linear objective function can be computed in pseudo-polynomial time, in $O(m(\sum_{i=1}^{n}|V_i|)^2 \log(\sum_{i=1}^{n}|V_i|))$ time.*

3.2.3. The generic construction. In order to motivate the construction of $G(\mathcal{L})$, we now present without proof how it can be obtained from the generic partial order $I(\mathcal{L})$.

Let $\mathcal{L}[x_i = a]$ denote the set of all feasible solution vectors for which $x_i = a$. Obviously, $\mathcal{L}[x_i = a]$ induces a sublattice of $\mathcal{L}$. We call a lattice element *irreducible* if for some variable x_i and integer a, it is the bottom element of the sublattice $\mathcal{L}[x_i = a]$. The partial order $(I(\mathcal{L}), \preceq)$ is defined as follows: the vertex set is the set of irreducible elements of the lattice $\mathcal{L}$; for elements $a, b \in I(\mathcal{L})$, there is an edge from a to b, if $a \preceq b$ in $\mathcal{L}$. The following theorem is proved in [6, p. 72, Thm. 9] and [7, Thm. 2.2.1].

THEOREM 3.8. *There is a one-to-one correspondence between the nonempty closed subsets of $I(\mathcal{L})$ and the elements of $\mathcal{L}$. Moreover, if closed subsets S and S' of $I(\mathcal{L})$ correspond to vectors L and L', respectively, then L' dominates L if and only if $S \subseteq S'$.*

However, the partial order $I(\mathcal{L})$ has a "complicated" structure which we now show how to simplify and make more regular. (This generalizes the construction in [10].)

The elements L_1 and L_2 are called *consecutive* elements in the lattice $\mathcal{L}$ if L_2 covers L_1, i.e., there is no element M such that $L_1 < M < L_2$. Suppose elements L_1 and L_2 are consecutive and $L_1 < L_2$. The *minimal difference* between L_1 and L_2 is defined to be the "set of changes" between L_1 and L_2. More formally, by a single *change* we mean the difference between the value of a variable in L_1 and L_2. We denote by $\mathcal{D}$ the set of all minimal differences in $\mathcal{P}$.

A *maximal chain* in a lattice is a chain of consecutive elements that starts at B and ends at T. An interesting property of distributive lattices is that each maximal chain contains *all* the minimal differences. The minimal differences appear on each maximal chain in some order and each minimal difference appears exactly once.

We can now define the partial order $(T(\mathcal{L}), \preceq)$. Let $D_1, D_2 \in \mathcal{D}$; then $D_1 < D_2$ if and only if D_1 precedes D_2 on every maximal chain in $\mathcal{P}$. We are now ready for the next theorem, whose proof follows from [7, Thm. 2.4.4] and which relates the partial orders $I(\mathcal{L})$ and $T(\mathcal{L})$.

THEOREM 3.9. *There is a one-to-one correspondence between the closed subsets of $I(\mathcal{L})$ and $T(\mathcal{L})$.*

In fact, the partial order $T(\mathcal{L})$ is very similar to $G(\mathcal{L})$. Let $\hat{V}_i$ denote the set of integer feasible values of variable x_i. Notice that the elements of $\hat{V}_i$ do not necessarily form a consecutive interval, in contrast to the fractional case, where all values λ such that $x_i^{\min} \leq \lambda \leq x_i^{\max}$ are feasible. Notice that $T(\mathcal{L})$ is the structure we obtain if we follow the definition of $G(\mathcal{L})$ except that the set V_i is replaced by $\hat{V}_i$. For example, if $\hat{V}_i = \{1, 5, 6\}$, then there is an arc from the vertex corresponding to "1" to the vertex corresponding to "5", and an arc from the vertex corresponding to "5" to the vertex corresponding to "6."

The difficulty in constructing the partial order $T(\mathcal{L})$ is that we need to generate the elements of the sets $\hat{V}_i$ one by one, since they are not necessarily sets of consecutive integers. This can be done in pseudo-polynomial time; however, $T(\mathcal{L})$ does not have a succinct description which motivates the construction of $G(\mathcal{L})$.

4. Identifying fat polytopes. This section presents an application of the Fourier–Motzkin algorithm for identifying *fat* polytopes.

Even though it is NP-hard to decide whether a set of inequalities has an integer feasible solution, one can use a fast preprocessing stage to compute an integer feasible solution in certain cases. This preprocessing stage runs in strongly polynomial time for the case of linear programs with two variables per inequality. It checks whether the polytope is *fat*, i.e., whether it contains a sphere circumscribing a unit hypercube. Since a unit hypercube must contain at least one integer lattice point, an integer feasible point is found by rounding the coordinates of the center of the sphere to the nearest integer. This procedure is a heuristic for finding a feasible integer point, since there may exist a feasible integer point in the polytope, yet the polytope does not contain a large enough sphere. Lenstra [12] uses a similar procedure that works in polynomial time and may identify a feasible integer point; however, in his procedure the running time depends on the ellipsoid method and is therefore not strongly polynomial.

The idea of the procedure is to shift all constraints by a distance of r. Any feasible point in the resulting set of inequalities is at a distance of r from all the faces of the polytope, and hence a sphere of radius r around any such feasible point is contained in the polytope. In order to obtain a sphere large enough to contain a unit hypercube, we need to set $r = \sqrt{n}/2$.

Shifting a constraint by a distance r is done as follows. Given an inequality $\sum_{i=1}^{n} a_i x_i \leq c$, the shifted inequality is $\sum_{i=1}^{n} a_i x_i \leq \bar{c}$, where

$$\bar{c} = c - \frac{r}{\sqrt{\sum_{i=1}^{n} a_i^2}} \sum_{i=1}^{n} a_i .$$

In the case of two variables per inequality, the sum includes at most two terms. The new set of inequalities, each with a constant c substituted by $\bar{c}$, is also a set of inequalities with two variables per inequality, and hence is solvable in the running time reported in §2. Consequently, we can test whether a polytope is *fat* and find an integer feasible point in $O(mn^2 \log m)$ time.

Although the running time is polynomial, the procedure involves the manipulation of square roots, which may be difficult in practice. Since this algorithm only finds a feasible integer point in a special case, it is satisfactory for this purpose to truncate $\bar{c}$ to a small number of accuracy bits, where the small number depends on the machine word length or on other implementation considerations.

An interesting related problem is to find the *largest* sphere contained in a polytope. For this we need to maximize the value of r such that the system still has a feasible solution. Although such a problem has three variables per inequality, it is still solvable in strongly polynomial time. This follows from the results of [3] and [13], where it is shown that a problem can be solved in strongly polynomial time if, by deleting a constant number of columns (in this case the constant is equal to one), it can be converted to a problem which is solvable in strongly polynomial time. Since treating r as a variable adds only one more variable to the problem, the problem of finding a largest sphere in the polytope defined by a set of inequalities with two variables per inequality is solvable in strongly polynomial time.

Acknowledgments. We thank the two anonymous referees for clarifying the presentation of the paper. We thank Arik Tamir for pointing out that the running time of the integer feasibility algorithm is pseudo-polynomial only in the case where the variables are bounded. We would also like to thank Edith Cohen for helpful remarks. Many thanks to Yossi Friedman for his help in making the figures.

REFERENCES

[1] B. Aspvall and Y. Shiloach, *Polynomial time algorithm for solving systems of linear inequalities with two variables per inequality*, SIAM J. Comput., 9 (1980), pp. 827–845.

[2] E. Cohen, *Combinatorial Algorithms For Optimization Problems*, Ph.D. thesis, Stanford Technical Report, STAN-CS-91-1366, June 1991.

[3] E. Cohen and N. Megiddo, *Strongly polynomial time and NC algorithms for detecting cycles in periodic graphs,* J. Assoc. Comput. Mach., 40 (1993), pp. 791–830.

[4] E. Cohen and N. Megiddo, *Improved algorithms for linear inequalities with two variables per inequality*, Proceedings of the Twenty Third Symposium on Theory of Computing, New Orleans, 1991, pp. 145–155. SIAM J. Comput., this issue, pp. 1313–1347.

[5] H. Edelsbrunner, G. Rote. and E. Welzl, *Testing the necklace condition for shortest tours and optimal factors in the plane*, Theoret. Comput. Sci., 66 (1989), pp. 157–180.

[6] G. Grätzer, *Lattice Theory: First Concepts and Distributive Lattices*, W. H. Freeman and Company, San Francisco, 1971.

[7] D. Gusfield and R. W. Irving, *The Stable Marriage Problem*, MIT Press, Cambridge, MA, 1989.

[8] A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[9] M. Iri, *Structural theory for the combinatorial systems characterized by submodular functions*, in Progress in Combinatorial Optimization, Academic Press, New York, 1984, pp. 197–219.

[10] S. Khuller, J. Naor. and P. N. Klein, *The lattice structure of flow in planar graphs*, SIAM J. Disc. Math., 6 (1993), pp. 477–490.

[11] J. C. Lagarias, *The computational complexity of simultaneous diophantine approximation problems*, SIAM J. Comput., 14 (1985), pp. 196–209.

[12] H. W. Lenstra Jr., *Integer programming with a fixed number of variables*, Math. of Oper. Res., 8 (1983), pp. 538–548.

[13] C. Haibt-Norton, S. Plotkin. and E. Tardos, *Using separation algorithms in fixed dimension*, Proceedings of the First Symposium On Discrete Algorithms, San Francisco, 1990, pp. 377–387.

[14] N. Megiddo, *Towards a genuinely polynomial algorithm for linear programming*, SIAM J. Comput., 12 (1983), pp. 347–353.

[15] C. G. Nelson, *An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem*, Technical Report AIM-319, Stanford University, 1978.

[16] J. C. Picard, *Maximal closure of a graph and applications to combinatorial problems*, Management Sci., 22 (1976), pp. 1268–1272.

[17] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.

[18] R. Shostak, *Deciding linear inequalities by computing loop residues*, J. Assoc. Comput. Mach., 28 (1981), pp. 769–779.

[19] D. Topkis, *Minimizing a submodular function on a lattice*, Oper. Res., 26 (1978), pp. 305–321.

[20] A. F. Veinott, *Representation of general and polyhedral subsemilattices and sublattices of product spaces*, Linear Algebra Appl., 114/115 (1989), pp. 681–704.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1193–1215, December 1994

COMPUTING THE ORDER OF A LOCALLY TESTABLE AUTOMATON*

SAM M. KIM[†] AND ROBERT MCNAUGHTON[‡]

Abstract. A locally testable language is a language with the property that, for some positive integer j, whether or not a string x is in the language depends on (1) the prefix and suffix of x of length $j-1$, and (2) the set of substrings of x of length j, without regard to the order in which these substrings occur or the number of times each substring occurs. For any j for which this is true, it is said that the language is *j-testable*. For a given locally testable language, the smallest such number j is called the *order* of the language. Locally testable languages are regular and therefore these concepts apply to the finite automata that recognize the languages. The authors show that computing the order of a given locally testable deterministic automaton is NP-hard and present a polynomial-time ϵ-approximation algorithm for computing it. In addition, an upper bound of $2n^2+1$ on the order of a locally testable automaton of n states is obtained, and the co-NP-completeness of the problem of whether, for a given j, a given deterministic automaton is j-testable is proven.

Key words. finite state automaton, regular language, local testability, algorithm, NP-hard

AMS subject classifications. 68Q25, 68Q45, 68Q68

1. Introduction.

The concept of local testability is rooted in the study of pattern recognition. It is best understood in terms of a kind of computational procedure used to classify a two-dimensional image: a window of relatively small size is moved around on the image and a record is made of the various attributes of the image that are detected by what is observed through the window. No record is kept of the order in which the attributes are observed and the positions of the image, where each attribute occurs, or how many times it occurs. We say that a classification on the possible images is *locally testable* if a decision about how the image is classified can be made simply on the basis of the set of attributes that occur. Certainly, some patterns involve global constraints and therefore cannot be recognized by local testing. Nevertheless, for many patterns, local testing is sufficient. In [8], local testability is discussed in terms of diameter-limited perceptrons.

The one-dimensional analogy to this concept has been well studied and is the subject of this paper. If we think of such an image as a character string, then the classification becomes a language. Formally, a locally testable language is defined as follows [2], [9].

DEFINITION 1.1. *Let Σ be a finite alphabet. For an integer $k \geq 0$ and $x \in \Sigma^*$ such that $|x| \geq k+1$, define $f_k(x)$ as the prefix ("front end") of x of length k, $t_k(x)$ as the suffix ("tail") of x of length k, and $I_{k+1}(x)$ as the set of ("intermediate") substrings of x of length $k+1$, i.e., $\{v | v \in \Sigma^{k+1}$ and $x = uvw$, for some $u, w \in \Sigma^*\}$. For $|x| \leq k$, $f_k(x) = t_k(x) = x$ and $I_{k+1} = \emptyset$ (the empty set). Let $SV_{k+1}(x)$ denote the triple $(f_k(x), I_{k+1}(x), t_k(x))$, called the* substring vector *of x. For $L \subseteq \Sigma^*$,*

(a) *L is 0-testable if and only if it is Σ^* or $\emptyset$.*

(b) *For an integer $k \geq 0$, L is $(k+1)$-testable if and only if, for all $x, y \in \Sigma^*$, $SV_{k+1}(x) = SV_{k+1}(y)$ implies that either both x and y are in L or neither is in L.*

(c) *An automaton is j-testable if it accepts a j-testable language.*

(d) *A language or an automaton is locally testable, if it is j-testable for some $j \geq 0$.*

*Received by the editors July 29, 1991; accepted for publicaton (in revised form) July 15, 1993. Partial support for this research was provided by National Science Foundation grants CDA-8805910 and CCR-9114725.

[†]Computer Science Department, Rensselear Polytechnic Institute, Troy, New York 12180. Current address, Department of Computer Engineering, Kyungpook National University, Taegu, Korea (kims@bh.kyungpook.ac.kr).

[‡]Computer Science Department, Rensselaer Polytechnic Institute, Troy, New York 18120 (mcnaught@cs.rpi.edu).

It is well known that all locally testable languages are regular. The only automata considered in this paper are DFAs (deterministic finite automata). Throughout this paper we let $M = (Q, \Sigma, \delta, q_{st}, F)$ be a DFA, with the understanding that Q is the set of states, Σ the input alphabet, δ the state-transition function, $q_{st} \in Q$ the initial state and $F \subseteq Q$ the set of final (or accepting) states. For any $q \in Q$ and $w \in \Sigma^*$, by $\delta(q, w)$ we mean the state that results when input w is applied to M in state q. For notation we follow [4].

Clearly, if a language or an automaton is j-testable, it is j'-testable for any $j' > j$. We say a locally testable language or automaton has *testability order* j (*order*, for brevity) if it is j-testable but not $(j-1)$-testable. In the early 1970s, these languages and their finite state automata were extensively investigated [2], [9], [10], [12]. However, there were several open problems concerning locally testable automata that did not yield to practical solutions. (1) Is a given deterministic finite automaton locally testable? (2) If it is locally testable, what is the order? (3) What is the largest order that a locally testable finite automaton with n states can have?

Brzozowski and Simon [2] introduced the following characterization theorem for locally testable automata. Much of the theoretical work in this paper is based on part (b) of this theorem.

THEOREM 1.2 [2]. *For any M*

(a) *M is locally testable if and only if, for all $q \in Q$, $x \in \Sigma^+$, $y, z \in \Sigma^*$, and $n \geq |Q|$,*
 (i) $\delta(q, x^n y x^n) = \delta(q, x^n y x^n y x^n)$, *and*
 (ii) $\delta(q, x^n y x^n z x^n) = \delta(q, x^n z x^n y x^n)$.

(b) *M is $(k+1)$-testable, if and only if, for all $q \in Q$, $y, z \in \Sigma^*$ and $x \in \Sigma^k$,*
 (i) $xy = zx$ *implies* $\delta(q, xy) = \delta(q, xyy)$, *and*
 (ii) $\delta(q, xyxzx) = \delta(q, xzxyx)$.

In Fig. 1, we illustrate situations in which part (b) of Theorem 1.2 is not true. The falsity of condition (i) is illustrated in parts (a) and (b) of that figure; the falsity of condition (ii) is illustrated in part (c). Thus, $\delta(p_0, xy) = q_m \neq r_m = \delta(p_0, xyy)$ in both (a) and (b), and $\delta(p_0, xyxzx) = r_m \neq r'_m = \delta(p_0, xzxyx)$ in (c).

In [2] and [10], another characterization of locally testable automata was introduced in terms of algebraic properties of the semigroups of locally testable automata. (The semigroup concept will not be used in this paper.) The papers suggested algorithms for problems (1) and (2), which simply check whether the semigroup of the given automaton satisfies the appropriate algebraic properties. However, both algorithms take exponential time in the worst case, since the size of the semigroup of a deterministic automaton can be exponential in the number of states of the automaton. The papers left open whether it was possible to find better algorithms for problems (1) and (2). There was an answer to problem (3) in [2], which was a bound equal to the size of the semigroup of the automaton, which is also exponential in the number of states in the worst case. The answer we give in the present paper (Theorem 2.8) is polynomial in the size of automaton. Recently, in [6] we gave a characterization of locally testable deterministic finite automata and introduced an O(n^2) time algorithm for problem (1), the local testability problem, where n is the number of states of the automaton. Using the abbreviation SCC to mean a maximal strongly connected component of a directed graph, i.e., a subgraph that is strongly connected and is not a proper subgraph of another strongly connected subgraph, we restate this characterization theorem.

THEOREM 1.3 [6]. *An automaton M is locally testable if and only if its state transition graph satisfies the following two properties:*

(a) *No* SCC *of the state transition graph of the automaton has a pair of distinct states p and q such that $\delta(p, x) = p$ and $\delta(q, x) = q$, for any $x \in \Sigma^+$.*

(b) *For every pair of* SCCs *m_1 and m_2 such that m_1 is an ancestor of m_2, either*

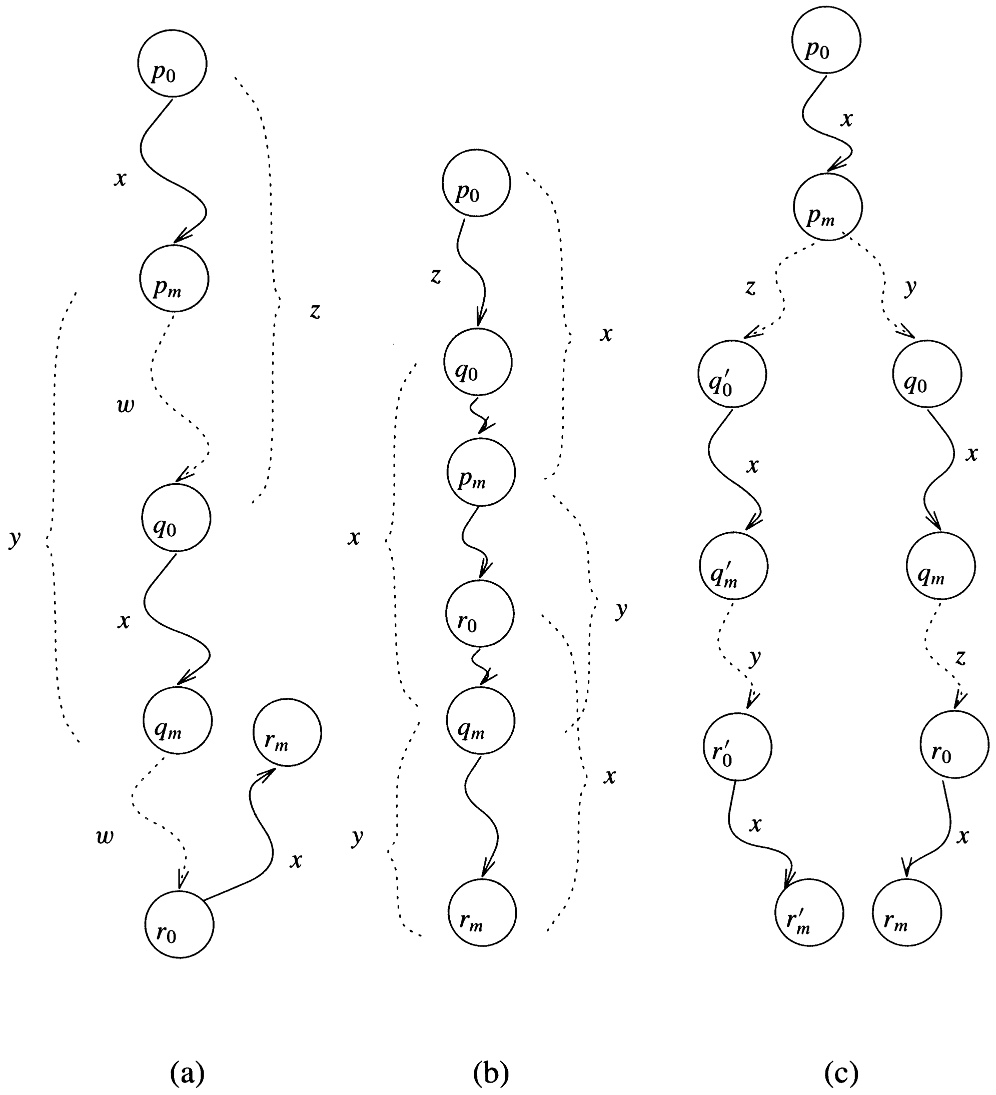


FIG. 1. *Typical transitions for part* (b) *of Theorem* 1.2.

(i) *no two states p in m_1 and q in m_2 exist such that $\delta(p, x) = p$ and $\delta(q, x) = q$, for any $x \in \Sigma^+$, or*

(ii) *for every such state pair p and q, there is a path from $\delta(p, w)$ to q if and only if $\delta(q, w)$ is in m_2, for every $w \in \Sigma^*$.*

This paper shows that problem (2), i.e., the problem of computing the order of a locally testable deterministic automaton, is NP-hard but has a polynomial-time ϵ-approximation algorithm. The algorithm, given a constant $\epsilon > 0$ and a locally testable deterministic automaton, computes an approximate order $\hat{k}$ of the automaton such that if k is the order then $\hat{k} \geq k$ and $(\hat{k} - k)/k \leq \epsilon$. For any constant ϵ the algorithm runs in polynomial time, the degree of the polynomial depending on ϵ. In order to deal with NP-hardness we define *the k-testability problem*: given a deterministic finite automaton M, and a nonnegative integer k, is M

k-testable? Following some preliminaries and a bound on the order in §2, §3 establishes that the k-testability problem is co-NP-complete. Sections 4 and 5 lay the theoretical groundwork for the approximation algorithm, §6 presents the algorithm itself and its analysis, and finally, §7 makes some concluding remarks.

2. Preliminaries and a bound on order. We assume throughout this paper that a problem instance is given in terms of the state transition graph of a reduced deterministic finite automaton M. This paper deals with edge-labeled directed graphs exclusively. The label on each edge is a letter of Σ. By *path* we mean a sequence of nodes $p_0 p_1 \dots p_n$ such that if p_{i+1} immediately follows p_i in the sequence then there is a directed edge from p_i to p_{i+1} in the graph. The length of the path is one less than the length of the sequence. The string $a_1 a_2 \dots a_n$ is a *span* of this path if and only if for each i, $1 \leq i \leq n$, a_i appears on the edge from p_{i-1} to p_i. For $p \in Q$, if $\delta(p, w) = p$, for some $w \in \Sigma^+$, then we say that p has a w loop.

We shall use q_e and q_d to denote, respectively, the accepting sink state and the nonaccepting sink state (or dead state) of the automaton; thus $\delta(q_e, a) = q_e$ and $\delta(q_d, a) = q_d$, for all $a \in \Sigma$.

It is trivial to identify 0-testable deterministic automata, i.e., those accepting Σ^* or $\emptyset$. Theorem 2.1 below shows a simple characterization of 1-testable deterministic automata, which can thereby be identified by a simple algorithm whose time complexity is on the order of the square of the size of the state transition graph.

Recalling Definition 1.1, note that $f_0(x) = t_0(x) = \epsilon$, for all strings x. Consequently, for all x and x', $SV_1(x) = SV_1(x')$ if and only if $I_1(x) = I_1(x')$. Thus a language L is 1-testable if and only if the following holds: for all x and x', if the set of letters occurring in x equals the set of letters occurring in x' then $x \in L$ if and only if $x' \in L$. In the proof of the following theorem, however, we do not use this characterization of 1-testability, but rather the characterization of part (b) of Theorem 1.2.

THEOREM 2.1. *A reduced automaton M is* 1*-testable if and only if, for all $q \in Q$ and $a, b \in \Sigma$,*

(1) *$\delta(q, a) = \delta(q, aa)$, and*

(2) *$\delta(q, ab) = \delta(q, ba)$.*

Proof. Assume first that M is 1-testable. We can show that in each case, since M is reduced, a violation of (1) or (2) would imply that the automaton is not 1-testable.

If $\delta(q, a) \neq \delta(q, aa)$, then we have $\delta(q, xax) \neq \delta(q, xaxax)$, for $x = \epsilon$. By letting $y = ax$ and $z = xa$, we have $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$. The automaton does not satisfy Condition (i) of Theorem 1.2(b) for x of length zero, and so it is not 1-testable.

If $\delta(q, ab) \neq \delta(q, ba)$ then, for $x = \epsilon$, we have $\delta(q, xaxbx) \neq \delta(q, xbxax)$. The automaton does not satisfy Condition (ii) of Theorem 1.2(b) for x of length zero, and so it is not 1-testable.

For the proof of sufficiency, suppose that M satisfies conditions (1) and (2) of our theorem. We note that, using mathematical induction on $|y|$, it can easily be proved that (2) implies

(a) for all $a \in \Sigma$, $y \in \Sigma^*$, $q \in Q$, $\delta(q, ay) = \delta(q, ya)$.

From (1) and (a), for any $u, v \in \Sigma^*$, we get $\delta(q, uav) = \delta(q, uaav) = \delta(q, uava)$. Hence, by taking y as uav, we get

(b) for all $y \in \Sigma^*$, $q \in Q$, if a is a letter in y then $\delta(q, y) = \delta(q, ya)$.

From (b) and (a), respectively, we get, for all $q \in Q$ and $y, z \in \Sigma^*$,

(c) $\delta(q, y) = \delta(q, yy)$ and

(d) $\delta(q, yz) = \delta(q, zy)$.

Finally, by part (b) of Theorem 1.2, (c) and (d) above imply that M is 1-testable. □

The test that Theorem 2.1 gives us is, as far as we know, the most efficient test to determine whether a given reduced deterministic finite automaton is 1-testable. We now proceed with the theoretical development needed to achieve our announced objectives for this section, which will not make use of Theorem 2.1.

We need the following well-known theorem (e.g., [3, p. 7]).

THEOREM 2.2. *For all strings x, y, z, $xy = zx$ if and only if there are strings u and v and integer $h \geq 0$ such that $z = uv$, $y = vu$ and $x = (uv)^h u$.*

DEFINITION 2.3 (periodic string). *Let $x = a_1 a_2 \ldots a_m$ be a string over a finite alphabet Σ. String x is* periodic *with period d, $1 \leq d \leq m - 1$, if $a_i = a_{i+d}$, for all i, $1 \leq i \leq m - d$.*

Notice that if a string x of length m is periodic with a period d, then $x = z^k u$, where z is the prefix of x of length d, $k = \lfloor m/d \rfloor$ and u is the prefix of z of length m mod d. The following lemma directly follows from Definition 2.3.

LEMMA 2.4. *If a string x is periodic with period d, then x has the following properties. Let z and y be, respectively, the prefix and suffix of length d of x.*

(a) *$x = zw = wy$, for some string w.*

(b) *$x = z^h u$, where $h = \lfloor |x|/d \rfloor$ and u is the prefix of length $|x|$* mod *d of z.*

(c) *every substring of length $> d$ of x is also periodic with the same period d.*

In order to establish that the k-testability problem is in co-NP we use the fact that a locally testable automaton M is k-testable for some $k \leq 2n^2 + 1$, where n is the number of states of M. This polynomial bound on k will be established in Theorem 2.8 below.

LEMMA 2.5. *Let M be reduced and locally testable. Let q_0 and q_1 be a pair of states such that $\delta(q_0, z) = q_0$ and $\delta(q_1, z) = q_1$ for some $z \in \Sigma^+$. If $\delta(q_0, w) = q_1$, then, for every $i \geq 0$ and any prefix u of w, all nodes in the path corresponding to $\delta(q_1, w^i u)$ are in the same* SCC *with q_1.*

Proof. Since q_0 has a loop and q_1 also has a loop, each must be in an SCC. If $q_0 = q_1$ the matter is trivial, so we assume $q_0 \neq q_1$. But then, since M is locally testable, q_0 and q_1 must be in distinct SCCs. Let m be the SCC which includes q_1, and for each $j \geq 1$, let $\delta(q_0, w^j) = q_j$.

Since $\delta(q_0, w) = q_1$ and there is a path from q_1 to q_1, by condition (ii) of Theorem 1.3(b), $\delta(q_1, w) = q_2$ is in m. Since $\delta(q_1, w) = \delta(q_0, w^2)$ is in m, by the same theorem, $\delta(q_1, w^2) = q_3$ is in m. Continuing in this way, we see that all the q_j's, for $j \geq 1$, are in m, as are all nodes in the path corresponding to $\delta(q_1, w^j)$. Thus, for any prefix u of w, all nodes in the path corresponding to $\delta(q_1, w^{j-1} u)$ are also in m. □

LEMMA 2.6. *Let M have n states (not necessarily reduced) and let $x \in \Sigma^*$, $|x| \geq n^2$. Then, for any states p and r, there exist x', w and x'' such that $x = x'wx''$, $|x'w| \leq n^2$, $|w| \geq 1$, $\delta(p, x'w) = \delta(p, x')$, and $\delta(r, x'w) = \delta(r, x')$ (viz., $\delta(p, x')$ and $\delta(r, x')$ both have a loop with the same span w).*

Proof. Let $|x| = k$, $p = p_0$ and $r = r_0$. For $0 \leq i \leq k$, let $\sigma_i = (\delta(p_0, t_i), \delta(r_0, t_i)) = (p_i, r_i)$, where t_i is the prefix of x of length i. Since $k \geq n^2$, by the pigeonhole principle there exist i_1, i_2, $0 \leq i_1 < i_2 \leq n^2 \leq k$, such that $\sigma_{i_1} = \sigma_{i_2}$, viz., $p_{i_1} = p_{i_2}$ and $r_{i_1} = r_{i_2}$. Set x', w and x'' so that $|x'| = i_1$, $|w| = i_2 - i_1$ and $x'wx'' = x$. Then the required equations are satisfied. □

LEMMA 2.7. *Let m be an* SCC *of a locally testable automaton, and let h be the number of states in m. For $u, v, x \in \Sigma^*$, $|x| \geq h^2$, and a pair of states s and t both in m, which are not necessarily distinct, if $\delta(s, ux)$ and $\delta(t, vx)$ are both in m, then $\delta(s, ux) = \delta(t, vx)$.*

Proof. Let M be the subautomaton of the given automaton whose state transition graph is m. Let $p = \delta(s, u)$ and $r = \delta(t, v)$. Since $|x| \geq h^2$, by Lemma 2.6 there exist x', w, and x'' such that $x = x'wx''$, $|x'w| \leq h^2$, $\delta(p, x') = \delta(p, x'w)$, and $\delta(r, x') = \delta(r, x'w)$. Thus, both $\delta(p, x')$ and $\delta(r, x')$ have a loop with span w. Since m is an SCC of a locally testable deterministic automaton, by condition (i) of Theorem 1.3(b), it must be $\delta(p, x') = \delta(r, x')$, which gives $\delta(p, x'wx'') = \delta(r, x'wx'')$, i.e., $\delta(p, x) = \delta(r, x)$. Thus $\delta(s, ux) = \delta(t, vx)$. □

THEOREM 2.8. *If M is a reduced locally testable finite automaton with n states, then M is $(2n^2+1)$-testable. In other words, the upper bound on the order of a locally testable automaton with n states is $2n^2+1$.*

Proof. For the proof by contradiction, suppose that M is not $(k+1)$-testable for some $k \geq 2n^2$. Then, by part (b) of Theorem 1.2, there exist $q \in Q$, $y, z \in \Sigma^*$ and $x \in \Sigma^k$, such that either $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$, or $\delta(q, xyxzx) \neq \delta(q, xzxyx)$.

Case I. $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$. By Theorem 2.2, there exist $u, v \in \Sigma^*$ such that $y = vu$, $z = uv$ and $x = (uv)^h u$, for some $h \geq 0$. Let $a = |y| = |z|$. Note that $a > 0$; consequently, $2n^2 < |xy| = k + a < |xyy| = k + 2a$.

Let $q_0 = q$ and, for each i, $1 \leq i \leq k+2a$, let $q_i = \delta(q_0, t_i)$, where t_i is the prefix of xyy of length i. Note that $xyy = zxy = zzx$, $\delta(q_0, z) = q_a$, $\delta(q_0, x) = q_k$, and $\delta(q_a, x) = q_{k+a}$. Applying Lemma 2.6 with x, q_0, and q_a, we find x', α, and x'' such that $x = x'\alpha x''$, $|x'\alpha| \leq n^2$, $|\alpha| \geq 1$, $\delta(q_0, x') = \delta(q_0, x'\alpha)$, and $\delta(q_a, x') = \delta(q_a, x'\alpha)$. Letting $i_1 = |x'|$, we have $\delta(q_{i_1}, \alpha) = q_{i_1}$ and $\delta(q_{a+i_1}, \alpha) = q_{a+i_1}$. We illustrate this in Fig. 2(a). Note the following, which we shall use later.

(a) $x''y = z'x''$, for some z' (i.e., x'' is a suffix of $x''y$), since $xy = x'\alpha x''y = zx'\alpha x''$.

(b) $|x''| \geq n^2$, since $|x| \geq 2n^2$ and $|x'\alpha| \leq n^2$.

(c) $\delta(q_0, zx'\alpha) = \delta(q_0, zx') = q_{i_1+a}$.

Since $x''yy$ is a substring of length $> |z|$ of the periodic string $x'\alpha x''yy$, by part (c) of Lemma 2.4 it is also a periodic with period a. Let β be the prefix of $x''yy$ of length a, so that $\delta(q_{i_1}, \beta) = q_{a+i_1}$. By part (b) of Lemma 2.4 we have $x''yy = \beta^{h'}u'$, for some $h' \geq 1$, where u' is a prefix of β.

We note that Lemma 2.5 can be applied as follows. Take α, β, q_{i_1}, and q_{i_1+a}, respectively, as the z, w, q_0, and q_1 of Lemma 2.5. We have $\delta(q_{i_1}, \alpha) = q_{i_1}$, $\delta(q_{i_1}, \beta) = q_{i_1+a}$, and $\delta(q_{i_1+a}, \alpha) = q_{i_1+a}$, which establishes the application. Let m be the SCC with q_{i_1+a}. From Lemma 2.5 we infer that all nodes on the path from q_{i_1+a} having the string $\beta^i u'$ (for any value of $i \geq 0$) as a span are in the same SCC m. But this set includes all the nodes on the path corresponding to the transition $\delta(q_{i_1+a}, x''y)$. Since $x''y = z'x''$ from (a) above, all the nodes in the transition $\delta(q_{i_1+a}, z'x'')$ are in SCC m.

Recalling that $|x''| \geq n^2$ from (b) above, we apply Lemma 2.7 with $s = q_{i_1+a}$, $t = q_{i_1+a}$, $u = \epsilon$, $v = z'$, and $x = x''$, and conclude that $\delta(q_{i_1+a}, x'') = \delta(q_{i_1+a}, z'x'')$. But from (c) above we get $\delta(q_{i_1+a}, x'') = \delta(q_0, zx'\alpha x'') = \delta(q_0, zx) = \delta(q_0, xy)$ and $\delta(q_{i_1+a}, z'x'') = \delta(q_{i_1+a}, x''y) = \delta(q_0, zx'\alpha x''y) = \delta(q_0, zxy) = \delta(q_0, xyy)$. It follows that $\delta(q_0, xy) = \delta(q_0, xyy)$, a contradiction.

Case II. $\delta(p_0, xyxzx) \neq \delta(p_0, xzxyx)$ (using p_0 in place of q). For $1 \leq i \leq k$, let $p_i = \delta(p_0, t_i)$ where t_i is the prefix of x of length i. Thus, $p_k = \delta(p_0, x)$. Let $\delta(p_k, y) = q_0$, $\delta(p_k, z) = q'_0$, $\delta(q_0, x) = q_k$, $\delta(q'_0, x) = q'_k$, $\delta(q_k, z) = r_0$, and $\delta(q'_k, y) = r'_0$. Finally, for $1 \leq i < k$, let q_i, q'_i, r_i, r'_i be, respectively, $\delta(q_0, t_i)$, $\delta(q'_0, t_i)$, $\delta(r_0, t_i)$, $\delta(r'_0, t_i)$, where t_i is the prefix of x of length i. Thus $r_k = \delta(p_0, xyxzx)$ and $r'_k = \delta(p_0, xzxyx)$. All this is summarized in Fig. 2(b).

Applying Lemma 2.6 with states p_0 and r_0 and string x, we find x', w, and x'' such that $x = x'wx''$, $|x'w| \leq n^2$, $|w| \geq 1$, $\delta(p_0, x') = \delta(p_0, x'w)$, and $\delta(r_0, x') = \delta(r_0, x'w)$. Letting $|x'| = i_1$, we have $\delta(p_{i_1}, w) = p_{i_1}$ and $\delta(r_{i_1}, w) = r_{i_1}$. Let C_p and C_r be the SCCs containing p_{i_1} and r_{i_1}, respectively.

By condition (ii) of Theorem 1.3(b), since there is a path from $\delta(p_{i_1}, x''yxzx'w^k) = r_{i_1}$ to r_{i_1}, the node $s = \delta(r_{i_1}, x''yxzx'w^k)$ is in C_r. Since $\delta(r_{i_1}, w^k) = r_{i_1}$ and $|w^k| > n^2$, we can apply Lemma 2.7, taking s and t of the lemma to be s and r_{i_1}, respectively; we infer $s = r_{i_1}$. Thus r_{i_1} has a $x''yxzx'w^k$ loop. Since $\delta(r_{i_1}, x'') = r_k$ is on this loop, it is in the SCC C_r.

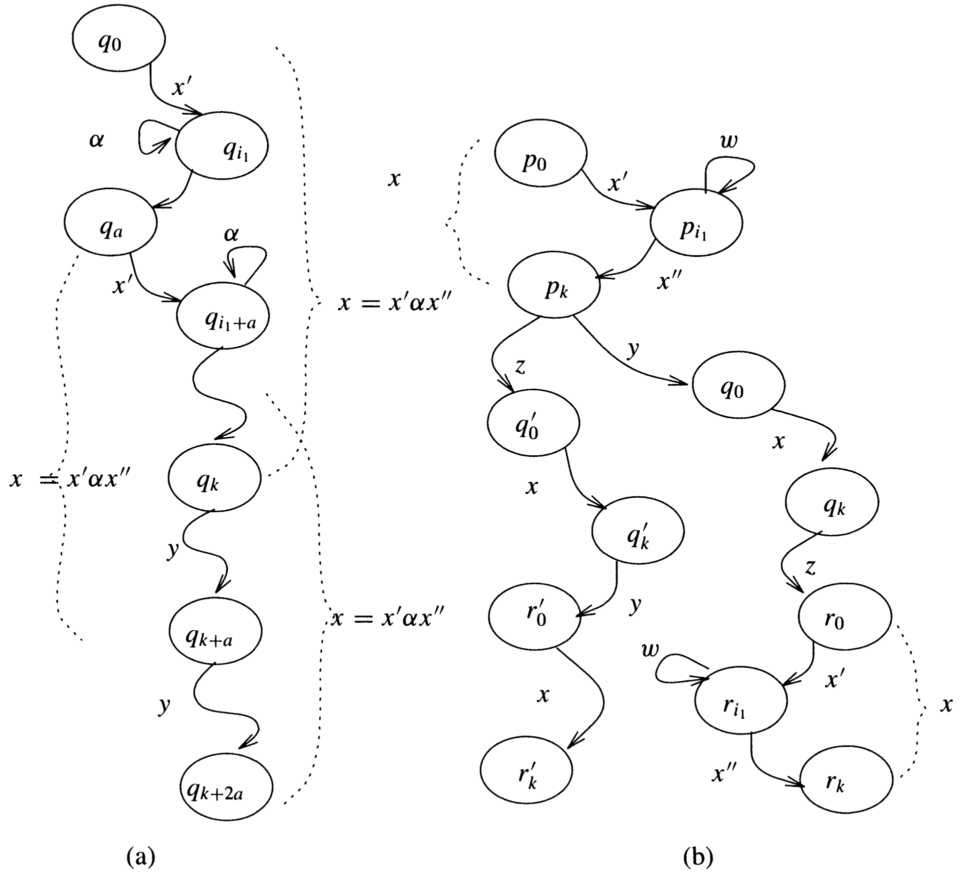


FIG. 2. *Proof of Theorem* 2.8. (a) *Case* I; (b) *Case* II.

Again, we apply condition (ii) of Theorem 1.3(b) with p_{i_1} and r_{i_1}. Since there is a path from $r_k = \delta(p_{i_1}, x''yxzx'wx'') = \delta(p_{i_1}, x''yxzx)$ to r_{i_1}, the node $s' = \delta(r_{i_1}, x''yxzx)$ is in C_r. Thus, we have $r_k = \delta(r_{i_1}, x'')$, $s' = \delta(r_{i_1}, x''yxzx)$, and $t = \delta(r_{i_1}, x''yx)$ all in SCC C_r. Since $x = x'wx''$ and $|x''| \geq n^2$, by Lemma 2.7, we have $r_k = s' = t$. It follows that $\delta(r_k, yx) = \delta(r_k, zx) = r_k$. Since $\delta(r_{i_1}, x''zxyx) = r_k$ is in SCC C_r, by condition (ii) of Theorem 1.3(b), there must be a path from $\delta(p_{i_1}, x''zxyx) = r'_k$ to r_k.

In summary, we have proved that (1) $\delta(r_k, yx) = r_k$, (2) $\delta(r_k, zx) = r_k$, and (3) there is a path from r'_k to r_k. By symmetry (interchanging y and z, q_i and q'_i, r_i and r'_i for each i) we can prove that (1) $\delta(r'_k, zx) = r'_k$, (2) $\delta(r'_k, yx) = r'_k$, and (3) there is a path from r_k to r'_k, which is in an SCC $C_{r'}$.

It follows that r_k and r'_k are in the same SCC, i.e., $C_r = C'_r$. Since $\delta(r_k, yx) = r_k$ and $\delta(r'_k, yx) = r'_k$, by Lemma 2.7 we get $r_k = r'_k$, a contradiction.

Both Case I and Case II having been proved contradictory; the proof of our theorem is complete. □

Since k-testability implies k'-testability for any $k' > k$, we have the following corrollary.

COROLLARY 2.9. *If M has n states and is reduced, then M is locally testable if and only if it is $(2n^2 + 1)$-testable.*

As far as we know $2n^2 + 1$ is the best upper bound on the order of a reduced locally testable DFA. We conjecture that the order is no greater than $O(n^{1.5})$ when the alphabet size is 2, and no less than $O(n^2)$ in the general case. Figure 3 shows an idea for constructing a locally testable automaton whose order approaches a bound that is $O(n^{1.5})$. The automaton has order 128 with 29 states, including the nonaccepting sink state q_d not shown in the figure. Let $x = a^{26}ba^{25}ba^{23}ba^{20}ba^{16}ba^{11}$, $y = bax$, and $z = xba$. Then $xy = zx$ and $\delta(p, xy) = t \neq \delta(p, xyy) = q_d$. It is not difficult to show that for any state q of the automaton and $x', y', z' \in \{a, b\}^+$ such that $x'y' = z'x'$ and $|x'| > |x|$, it satisfies $\delta(q, x'y') = \delta(q, x'y'y') = q_d$ and $\delta(q, x'y'x'z'x') = \delta(q, x'z'x'y'x') = q_d$. Since $|x| = 126$, by part (b) of Theorem 1.2 the automaton is not 127-testable, but we are satisfied that there is no similar situation with any x of length 127. Thus the order is 128. We think that this is the largest order achievable with 29 states and two input symbols. (In [6] we discussed this example, stating erroneously that the automaton had 28 states and had order 127.)

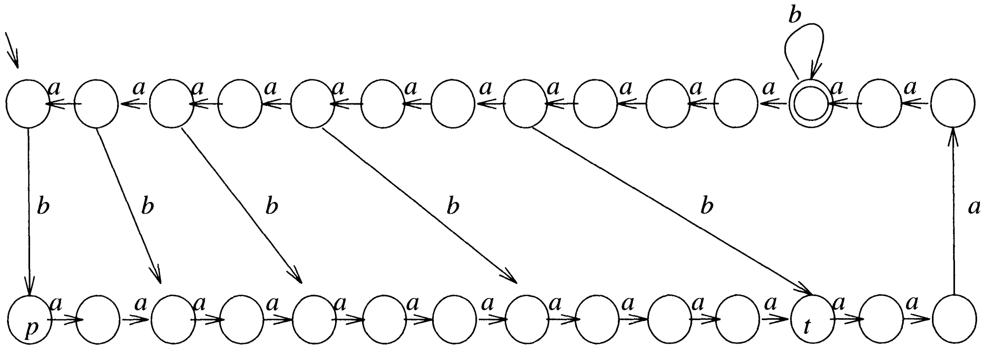


FIG. 3. *A* DFA *whose order is* 128 *with* 29 *states.*

3. Proof of co-NP-completeness.

With the groundwork established in §2, this section shows that the problem of computing the order of a locally testable deterministic automaton is NP-hard by proving that the k-testability problem is co-NP-complete. Before we show the main result, we need some working lemmas.

LEMMA 3.1. *If M has n states and y is a string such that $\delta(p_1, y) = q_1$ and $\delta(p_2, y) = q_2$, then there is a y_1 such that $|y_1| \leq n^2 - 1$, $\delta(p_1, y_1) = q_1$, and $\delta(p_2, y_1) = q_2$.*

Proof. We prove the lemma by proving instead that if y satisfies the hypothesis but does not satisfy $|y| \leq n^2 - 1$, then there is a shorter y_1 satisfying the hypothesis.

So, suppose $|y| \geq n^2$. Then, by Lemma 2.6, there exist y', w, and y'' such that $y = y'wy''$, $|y'w| \leq n^2$, and $|w| \geq 1$, and both $\delta(p_1, y')$ and $\delta(p_2, y')$ have w loops. It follows that $\delta(p_1, y'y'') = q_1$ and $\delta(p_2, y'y'') = q_2$. Then $y_1 = y'y''$ satisfies the hypothesis. □

LEMMA 3.2. *Let M have n states and let $k \geq 1$, $q \in Q$, $x \in \Sigma^{k-1}$, and $y, z \in \Sigma^*$. If $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$, then there exist y', z' such that $|y'|, |z'| \leq k + n^2 - 2$, $xy' = z'x$, and $\delta(q, xy') \neq \delta(q, xy'y')$.*

Proof. Clearly, $|y| = |z|$. Assume that $|y| > k + n^2 - 2$. Then $|y| > |x|$ and the equation $zx = xy$ implies the existence of y_1 such that $z = xy_1$ and $y = y_1x$. Hence, $xy = xy_1x$ and $xyy = xy_1xy_1x$. Let $\delta(q, x) = q_1$, $\delta(q_1, y_1) = q_2$, $\delta(q_2, x) = q_3$, $\delta(q_3, y_1) = q_4$, and $\delta(q_4, x) = q_5 \neq q_3$.

Applying Lemma 3.1 with states q_1 and q_3, and string y_1, we find y_1', $|y_1'| \leq n^2 - 1$, such that $\delta(q_1, y_1') = q_2$ and $\delta(q_3, y_1') = q_4$. Putting $y' = y_1'x$ and $z' = xy_1'$ we get $|y'| = |z'| \leq k + n^2 - 2$, $z'x = xy'$, and $\delta(q, xy') = q_3 \neq q_5 = \delta(q, xy'y')$. □

LEMMA 3.3. *Let M have n states, $q \in Q$, and $x, y, z \in \Sigma^*$. If $\delta(q, xyxzx) \neq \delta(q, xzxyx)$ then, for some y', z' where $|y'|, |z'| \leq n^2 - 1$, $\delta(q, xy'xz'x) \neq \delta(q, xz'xy'x)$.*

We omit the proof, which makes straightforward use of Lemma 3.1.

THEOREM 3.4. *The k-testability problem is in co-NP.*

Proof. For given deterministic automaton M and nonnegative integer k, the nondeterministic algorithm for the co-problem of the k-testability problem is shown in Fig. 4. This polynomial time procedure will terminate negative without reaching step 3 or will terminate negative for some allowable choice in step 3 if and only if M is not k-testable. We leave the proof to the reader, noting only that Theorem 1.2(b), Theorem 2.8, and Lemmas 3.2 and 3.3 are involved; in particular, the two lemmas justify restricting the choice of n_1 and n_2 to integers less than $k + n^2 - 2$. □

```
Procedure CO-NP ($M$, $k$);
(// $n$ is the number of states of $M$ //)

begin
1. if $k = 0$ then
      begin
         if $M$ is 0-testable
                  then output "$M$ is $k$-testable" and terminate
                  else output "$M$ is not $k$-testable" and terminate
      end;

2. Nondeterministically choose $x \in \Sigma^{k-1}$, $q \in Q$, nonnegative integers
      $n_1, n_2 \leq k + n^2 - 2$ and strings $y \in \Sigma^{n_1}$ and $z \in \Sigma^{n_2}$;
3. if $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$
      then output "$M$ is not $k$-testable"
      else if $\delta(q, xyxzx) \neq \delta(q, xzxyx)$ then output "$M$ is not $k$-testable"
end.
```

FIG. 4. *The nondeterministic algorithm for the co-k-testability problem.*

Our proof of co-NP-completeness involves reducing SAT (the satisfiability problem of a propositional formula in conjunctive normal form) to the complement of the k-testability problem in polynomial time.

Let $A = C_1C_2 \ldots C_m$, $m \geq 4$, be a propositional formula in conjunctive normal form (CNF) having $r \geq 4$ variables $v_1, v_2, \ldots, v_r$. We shall show how to construct a locally testable deterministic automaton M that is not $5rm$-testable if and only if A is satisfiable. The idea is best understood by an example; Fig. 5 depicts the automaton for $A = C_1C_2C_3C_4$, where

$$\begin{aligned} C_1 &= v_1 \vee \overline{v_2} \vee \overline{v_4} \vee v_5, \\ C_2 &= v_1 \vee \overline{v_2} \vee v_4 \vee \overline{v_5}, \\ C_3 &= \overline{v_3} \vee \overline{v_5}, \\ C_4 &= \overline{v_1} \vee \overline{v_2} \vee v_4. \end{aligned}$$

The figure does not show the rejecting sink state q_d or the transitions to it.

In general, we construct the automaton $M = (Q, \{a, b\}, \delta, q_{st}, F)$ from A as follows. The automaton has the nonaccepting sink state q_d and the accepting sink state q_e. For each clause C_i, the state transition graph has two parallel paths $p_{i,0}p_{i,1} \ldots p_{i,r-1}$ and $q_{i,1}q_{i,2} \ldots q_{i,r}$, each having r states. Among these states δ is defined as follows. For all j, $0 \leq j \leq r - 1$, if C_i has variable v_{j+1} then $\delta(p_{i,j}, b) = q_{i,j+1}$ and $\delta(p_{i,j}, a) = p_{i,j+1}$ or q_d if $j = r - 1$. If C_i has $\overline{v_{j+1}}$ then $\delta(p_{i,j}, a) = q_{i,j+1}$ and $\delta(p_{i,j}, b) = p_{i,j+1}$ or q_d if $j = r - 1$. But if C_i has neither v_{j+1} nor $\overline{v_{j+1}}$ then $\delta(p_{i,j}, a) = \delta(p_{i,j}, b) = p_{i,j+1}$ or q_d if $j = r - 1$. For all j, $1 \leq j \leq r - 1$, $\delta(q_{i,j}, a) = \delta(q_{i,j}, b) = q_{i,j+1}$. We call the resulting graph G_i the clause-graph (CG) for C_i.

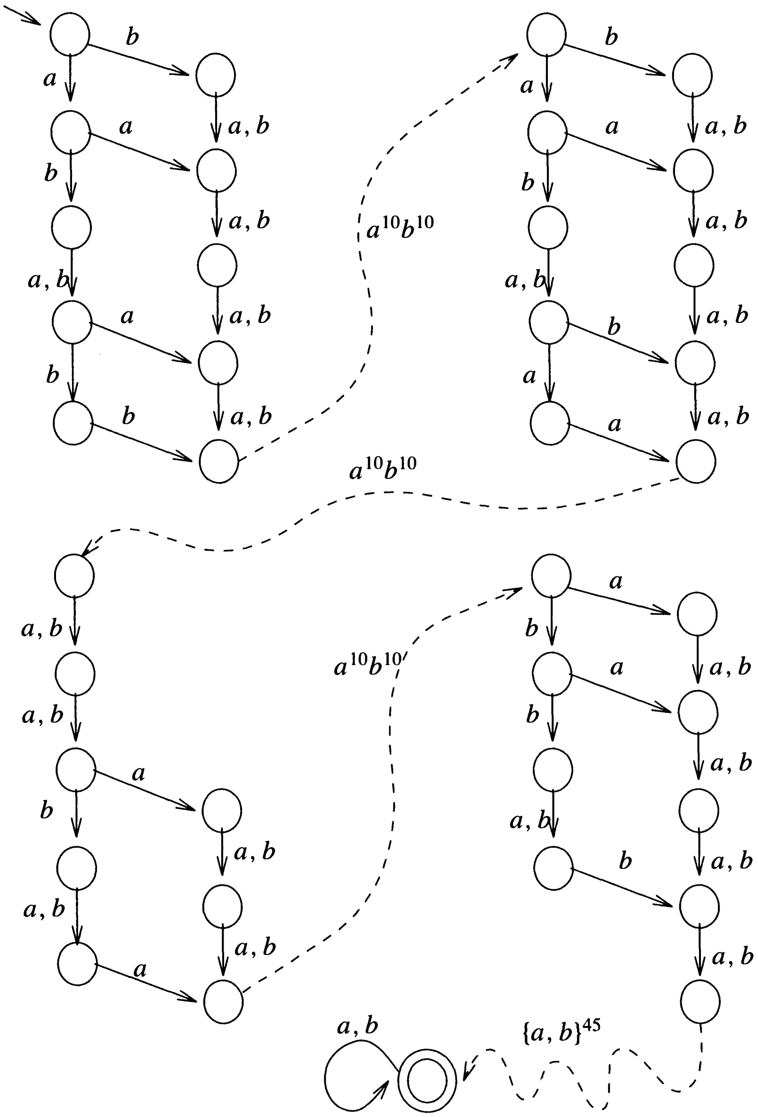


FIG. 5. *Constructing a locally testable* DFA *from a formula.*

Now, we link all CGs into a single chain by establishing a path with span $\alpha = a^{2r}b^{2r}$ from $q_{i,r}$ to $p_{i+1,0}$, $1 \leq i < m$, and we link $q_{m,r}$ of G_m to the accepting sink state q_e by a nonbranching path of length $2|\alpha| + r = 9r$. To say a path $P = q_1q_2 \ldots q_n$ is nonbranching is to say that for all i, $1 \leq i \leq n-1$, $\delta(q_i, a) = \delta(q_i, b) = q_{i+1}$. (The reason for the string α will be made clear in the following lemmas.) All transitions not so far defined go to the dead state q_d. Finally, we complete the construction by defining the start state q_{st} to be $p_{1,0}$ and reducing the automaton. Notice that in the reduced automaton q_e is the only accepting state, there is a path from the initial state to every other state, and there is a path from every state, except q_d, to q_e.

Let us make some observations about M and the language L that it accepts. Since every path from $p_{1,0}$ to q_e has length $5r(m+1)$ and q_e is the only accepting state, we have $L \subseteq \{a,b\}^{5r(m+1)}\{a,b\}^*$. There is only one path from $q_{m,r}$ to q_e, and all nodes on this path are nonbranching; this implies that, whenever $|z| = |z'|$, $\delta(q_{m,r}, z) = \delta(q_{m,r}, z')$. Finally, we note that the graph is acyclic except for q_e and q_d. Clearly, by Theorem 1.3, M is a locally testable automaton.

Now let $t_1, \ldots, t_r$ be truth value assignments, respectively, to variables $v_1, \ldots, v_r$, and let $w \in \Sigma^r$ be the string $\sigma_1\sigma_2 \ldots \sigma_r$, where, for each j, $\sigma_j = b$ if t_j is true, and $\sigma_j = a$ if t_j is false. Observe that there is a path in G_i from node $p_{i,0}$ to node $q_{i,r}$ with span w if and only if the clause C_i is true under this assignment. Therefore, the formula $C_1C_2 \ldots C_m$ is satisfiable if and only if there exists a string w such that $(w\alpha)^{m+1}$ is in the language of M. In Lemmas 3.5 through 3.8 below, M refers to the automaton that has just been constructed and $\alpha = a^{2r}b^{2r}$.

LEMMA 3.5. *Let $w_1, w_3 \in \Sigma^*$, $w_2 \in \Sigma^r$, and $x = w_1\alpha w_2\alpha w_3$. If x has period d, then $d \geq 5r$.*

Proof. Let $s = |w_1|$ and let $x[i]$, $1 \leq i \leq |x|$, denote the ith symbol in x. Then we have the following:

(a) $x[i] = a$ for all i, $s+1 \leq i \leq s+2r$, and for all i, $s+5r+1 \leq i \leq s+7r$.

(b) $x[i] = b$ for all i, $s+2r+1 \leq i \leq s+4r$, and for all i, $s+7r+1 \leq i \leq s+9r$.

If x has period d then $x[i] = x[i+d]$, for all i, $1 \leq i \leq |x| - d$. We prove our lemma by exhibiting a contradiction for each d, $1 \leq d \leq 5r-1$.

Case 1. $1 \leq d \leq 2r$. Then $x[s+2r] = a$ but $x[s+2r+d] = b$.

Case 2. $2r \leq d \leq 4r-1$. Then $x[s+1] = a$ but $x[s+1+d] = b$.

Case 3. $4r \leq d \leq 5r-1$. Then $x[s+2r+1] = b$ but $x[s+2r+1+d] = a$. □

LEMMA 3.6. *For $x \in \Sigma^*$, if $|x| \geq 5rm - 1$ then, for all $z, y \in \Sigma^*$ and $q \in Q$, $\delta(q, xyxzx) = \delta(q, xzxyx) \in M$.*

Proof. If $|x| \geq 5rm - 1$, by the construction of M, $\delta(q, x)$ is either q_d or a node on the nonbranching path from $q_{m,r}$ to q_e. (Indeed, this would be true even if $|x|$ were as small as $5rm - 4r$.) Thus, for all z_1 and z_2, $|z_1| = |z_2|$ implies $\delta(q, xz_1) = \delta(q, xz_2)$, which gives us the lemma. □

The following lemma is immediate from Theorem 1.2(b) and Lemma 3.6.

LEMMA 3.7. *If M is not $5rm$-testable, then there are q, x, y, z such that $|x| = 5rm - 1$, $xy = zx$, and $\delta(q, xy) \neq \delta(q, xyy)$.*

LEMMA 3.8. *The automaton M is not $5rm$-testable if and only if the formula $C_1 \ldots C_m$ is satisfiable.*

Proof. Assume M is not $5rm$-testable, and let q, x, y, z be the result of applying Lemma 3.7. The construction of M and the fact that $xy = zx$ and $\delta(q, xy) \neq \delta(q, xyy)$ imply that $q_d \neq \delta(q, xy) \neq q_e$ and $|xy| < 5r(m+1)$. Since $|x| = 5rm - 1$, we get $|y| \leq 5r$.

By construction of M, since $m \geq 4$ and $|x| = 5rm - 1$, xy must cover at least two α's, and therefore must be of the form $w_1\alpha w_2\alpha w_3$, where $|w_2| = r$. By Lemma 3.5, xy cannot have a period $< 5r$. By Theorem 2.2, $xy = zx$ implies that, for some u, v and $h \geq 0$, $y = vu$, $z = uv$ and $x = (uv)^h u$; hence, $xy = (uv)^{h+1}u$. Since $|x| \neq 0$, we cannot have both $|u| = 0$ and $h = 0$. Thus xy is periodic with period $|z|$. Thus $|y| = |z| \geq 5r$.

From the above conclusion, i.e., $|y| \leq 5r$, we get $|y| = 5r$, and hence

$$|xy| = 5r(m+1) - 1.$$

By the construction of M, the start state q_{st} is the only state from which there is a path of length $5r(m+1) - 1$ to a state other than q_e or q_d. Since $q_d \neq \delta(q, xy) \neq q_e$, we get $q = q_{st}$.

Since xy is a periodic span of a path of length $5r(m+1)-1$ with period $5r$ that starts from q_st, it must be that $xy = (w\alpha)^m wa^{2r}b^{2r-1}$ for some $w \in \Sigma^r$, where $\alpha = a^{2r}b^{2r}$. Thus w represents a set of truth values satisfying each of the clauses $C_1, C_2, \ldots, C_m$ and hence the formula is satisfiable.

It remains to prove that if the formula is satisfiable then M is not $5rm$-testable. But if it is satisfiable then there is a $w \in \Sigma^r$ that represents a satisfying set of truth values for every clause, implying that $\delta(q_{st}, (w\alpha)^{m+1}) = q_e$. Putting $x = (w\alpha)^{m-1}wa^{2r}b^{2r-1}$, $y = bwa^{2r}b^{2r-1}$, and $z = w\alpha$, we get $zx = xy$ and $\delta(q_{st}, xy) \neq q_e = \delta(q_{st}, xyy)$, showing that M is not $5rm$-testable. □

THEOREM 3.9. *The k-testability problem is co-NP-complete.*

Proof. Given a conjunctive normal formula with at least four clauses and at least four variables, we have shown how to construct an automaton M and an integer k such that M is not k-testable if and only if the formula is satisfiable. We leave it to the reader to verify that this construction can be completed in polynomial time. Thus SAT is polynomial-time reducible to the complement of the k-testability problem. From this and Theorem 3.4, the k-testability problem is co-NP-complete. □

COROLLARY 3.10. *The problem of computing the order of a locally testable automaton is NP-hard.*

4. Toward an approximation algorithm. Having shown that the problem of computing the order of a locally testable DFA is NP-hard, we seek to develop a polynomial-time algorithm for computing an approximate order. When we know a language L is locally testable, to make use of this fact we need to find a k such that L is k-testable, and we would like k to be as small as we can get it. The order k_{or} of L is the ideal value of k, since it is the smallest k such that L is k-testable. Since (probably) no tractable algorithm exists to find k_{or}, we must settle for some k that is larger and reasonably close. In §6 we shall present an ϵ-approximation algorithm that will serve this purpose: given ϵ and a locally testable automaton, it finds a value of $k \geq k_{or}$ such that

$$\frac{k - k_{or}}{k_{or}} \leq \epsilon.$$

(See [5] for other polynomial-time ϵ-approximation algorithms for NP-hard problems.) In this section and the next, we shall lay the theoretical foundation and a computational tool for our algorithm.

Let us reflect on part (b) of the Brzozowski–Simon theorem (Theorem 1.2). Note that if conditions (i) and (ii) are satisfied for all $x \in \Sigma^k$ then they are satisfied for any x of longer length. (In other words, if M is $(k+1)$-testable then it is $(k+i)$-testable for any $i > 1$.) Likewise if either (i) or (ii) fails for any $x \in \Sigma^{k-1}$ then it fails for some x of any shorter length. (If M is not k-testable then it is not $(k-i)$-testable for any $i > 0$.)

Thus a correct but inefficient algorithm to find the order of a given M, already known to be locally testable, is as follows: find the smallest k such that, for all $x \in \Sigma^k$, conditions (i) and (ii) are satisfied for all $q \in Q$, and $y, z \in \Sigma^*$, where $|y|, |z| \leq n^2 - 1$, n being the number of states of M. The order is then $k+1$. The limits on y and z are justified by Lemma 3.1. We know by Theorem 2.8 that we shall not have to search beyond $k = 2n^2 - 1$.

This algorithm takes exponential time, and our NP-hardness result inhibits any attempt to improve on it. However, we note that there is a polynomial-time algorithm to find the smallest k satisfying condition (ii). Since this algorithm is a part of our approximation algorithm, we shall present it (implicitly) in this section, deferring condition (i) to the next section.

DEFINITION 4.1 (n-tuple-graph, etc.). *For $n \geq 1$, the n-tuple-graph of M is the edge-labeled directed graph $G_n = G(V, E)$, where $V = Q^n$ and E is the set of labeled edges* *defined as follows. There is an edge from* $(p_1, p_2, \ldots, p_n)$ *to* $(q_1, q_2, \ldots, q_n)$ *labeled by* $a \in \Sigma$ *if* $\delta(p_i, a) = q_i$, *for all* i, $1 \leq i \leq n$. *Where* α *is a set of nodes of* G_n, *we obtain an abridgment* G_n^α *of* G_n *by deleting all the nodes of* α *and deleting all edges incident on nodes of* α. *Specifically, we define* $H_n = G_n^\alpha$, *where* α *is the set of all nodes* $(p_1, \ldots, p_{n-1}, p_n)$ *such that* $p_{n-1} = p_n$; $I_n = G_n^\beta$, *where* β *is the set of all nodes* $(p_1, \ldots, p_n)$ *such that, for some* i, j, $1 \leq i < j \leq n$, $p_i = p_j$; $H_n^{ac} = (H_n)^\gamma$, *where* γ *is the set of all nodes of* H_n *that have a loop; and* $I_n^{ac} = (I_n)^\eta$, *where* η *is the set of all nodes of* I_n *that have a loop.*

Note that a node $(p_1, \ldots, p_n)$ of G_n is in H_n if and only if $p_{n-1} \neq p_n$, and is in I_n if and only if $p_i \neq p_j$ for all i and j such that $i \neq j$. Note also that H_n^{ac} and I_n^{ac} are acyclic directed graphs (hence the designation "ac"). We shall use the graphs H_n, I_n, H_n^{ac}, and I_n^{ac} for various values of n in our algorithm. Obviously, $H_1 = I_1 = G_1$, which is the state transition graph of M itself; also $H_2 = I_2$. Figure 6 is an example of M, as given by its state transition graph, and the corresponding $H_2 = I_2$. Similarly, Fig. 7 is another M and its I_3 (which in this example happens to be the same as I_3^{ac}), showing only one sixth of the graph I_3; the other five portions of I_3 are each isomorphic to the part shown, with permuted node labeling (e.g., the permuted nodes corresponding to the visible node pqr are prq, qpr, qrp, rpq, and rqp). An example of a node in the H_3 of this M but not in I_3 is dds, to which there is a transition from the node dsr which is in I_3.

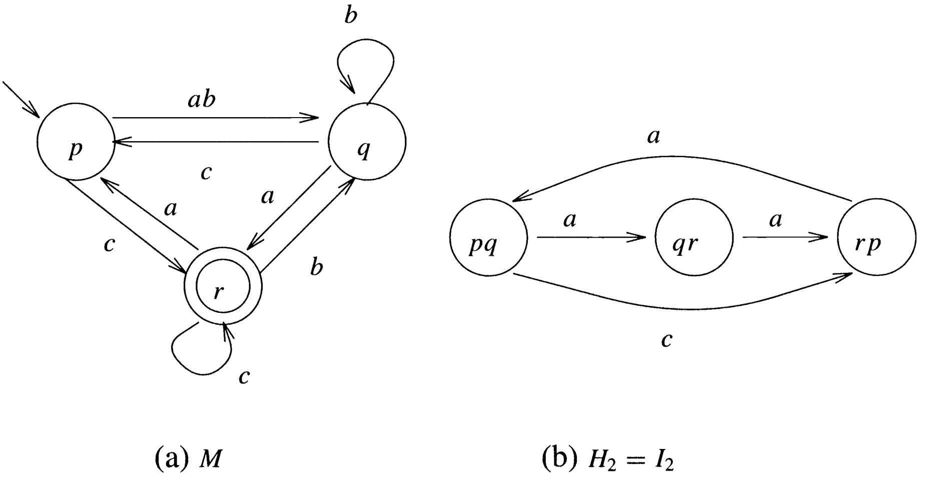


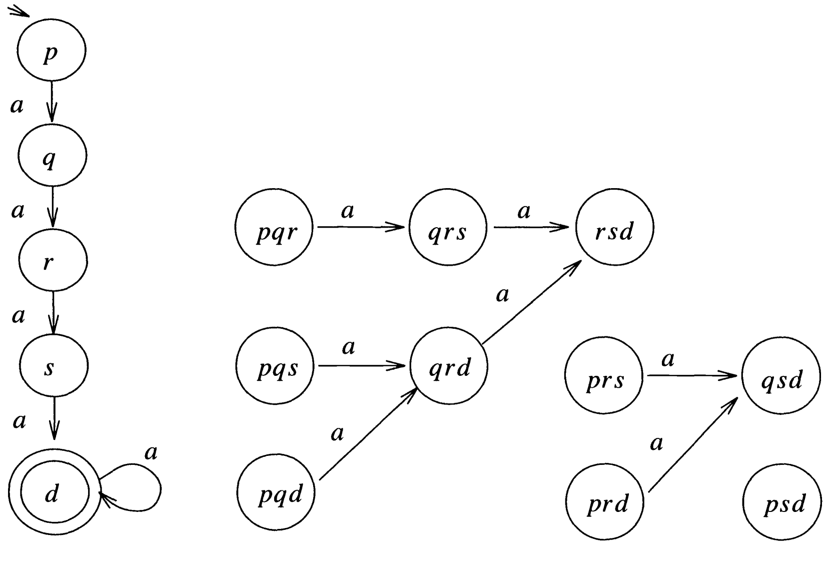
FIG. 6. *Constructing* H_2 *and* I_2 *for an automaton* M.

DEFINITION 4.2 (QCP). *Given* M, *let* $P = N_0 N_1 \ldots N_m$, $m \geq 0$, *be a path on* H_5, *where* $N_i = (p_i, q_i, q_i', r_i, r_i')$, $p_i, q_i, q_i', r_i, r_i' \in Q$, $1 \leq i \leq m$. *We say* P *is a* QCP (quintuple coherent-path) *if* G_2 *has a path from* (p_m, q_m) *to* (q_0', r_0) *and a path from* (p_m, q_m') *to* (q_0, r_0'). (*See Fig.* 1(c) *for a typical structure of transitions that give a* QCP *with span* x.)

For a graph G, we identify $|G|$ (the size of G) with the number of vertices plus the number of edges. For a finite automaton M, we identify $|M|$ with $i(j+1)$ where $i = |Q|$ and $j = |\Sigma|$. Thus the size of M is the size of the state transition graph G_1 of M, since i is the number of nodes and ij the number of edges of G_1.

LEMMA 4.3. *The graphs* G_n, H_n, *and* I_n *can be constructed from* M *and from* G_{n-1}, H_{n-1}, *and* I_{n-1}, *respectively, in time* $O(|M|^n)$.

Proof. Assume $|Q| = i$ and $|\Sigma| = j$. Since G_n has i^n nodes and each node has j outgoing edges, G_n can be constructed in time $O(ji^n)$, which is also a bound on the time for



(a) M (b) Part of I_3

FIG. 7. *Constructing I_3 for an automaton M.*

computing H_n and I_n. Since $|M|$ is of size $i(j+1)$, ji^n is $O(|M|^n)$. The same argument holds for the construction of H_n from H_{n-1} and the construction of I_n from I_{n-1}. □

LEMMA 4.4. *If M is locally a testable automaton and k is the largest integer such that H_5 has a* QCP *of length $k-1$ then k is the smallest integer satisfying condition* (ii) *of Theorem* 1.2(b), *i.e., for all $x \in \Sigma^k$, $q \in Q$, $y, z \in \Sigma^*$, $\delta(q, xyxzx) = \delta(q, xzxyx)$.*

Proof. Assume k is the largest integer such that H_5 has a QCP of length $k-1$; let this QCP be $P = N_0N_1 \ldots N_{k-1}$, $N_i = (p_i, q_i, q_i', r_i, r_i')$. Let $x \in \Sigma^{k-1}$ be a span of this path. By Definition 4.2 G_2 has paths from (p_{k-1}, q_{k-1}') to (q_0, r_0') and from (p_{k-1}, q_{k-1}) to (q_0', r_0). Let y be a span on the former path and z be a span on the latter.

Since N_{k-1} is a node of H_5, by Definition 4.1 we have $r_{k-1}' \neq r_{k-1}$. From the definition of QCP (see Fig. 1(c)) we conclude that $\delta(p_0, xyxzx) = r_{k-1} \neq r_{k-1}' = \delta(p_0, xzxyx)$. Thus $k-1$ does not satisfy condition (ii) of Theorem 1.2(b).

It remains to prove that k satisfies condition (ii). To this end, let $x \in \Sigma^k$, $y, z \in \Sigma^*$, and $p_0 \in Q$. Let $p_k = \delta(p_0, x)$, $q_0 = \delta(p_k, y)$, $q_0' = \delta(p_k, z)$, $q_k' = \delta(q_0', x)$, $q_k = \delta(q_0, x)$, $r_0 = \delta(q_k, z)$, $r_0' = \delta(q_k', y)$, $r_k = \delta(r_0, x)$, and $r_k' = \delta(r_0', x)$. For each i, $1 \leq i \leq k$, let x_i be the prefix of x of length i, $r_i = \delta(r_0, x_i)$, and $r_i' = \delta(r_0', x_i)$ as in Fig. 1(c).

We claim $r_k = r_k'$. To justify our claim we note that $r_k \neq r_k'$ implies $r_i \neq r_i'$, for each i. Thus $r_k \neq r_k'$ implies that the sequence $(p_0, q_0, q_0', r_0, r_0'), \ldots, (p_k, q_k, q_k', r_k, r_k')$ is a path in H_5 and is, therefore, a QCP of length k, whose connecting paths are as indicated in the paragraph above. This contradicts the hypothesis of our lemma.

Thus $r_k = r_k'$ and we are able to infer that $\delta(p_0, xyxzx) = \delta(p_0, xzxyx)$. The node p_0 and $x \in \Sigma^k$, $y, z \in \Sigma^*$ being arbitrary, we conclude that k satisfies condition (ii) of Theorem 1.2(b). □

LEMMA 4.5. *No* QCP *of a locally testable* DFA *contains a node that has a loop in H_5. Thus all* QCPs *are in H_5^{ac}.*

Proof. Suppose node N_i, $1 \leq i \leq m$, of a QCP $N_0 \ldots N_i \ldots N_m$ has a loop, i.e., there is a path $N_0' \ldots N_j'$, $j \geq 1$, such that $N_0' = N_j' = N_i$. Clearly, for any positive integer α, the path $N_0 \ldots (N_0' \ldots N_j')^\alpha \ldots N_m$, is also a QCP of length $m + (\alpha - 1)j$.

Since there are QCPs of arbitrary length, from Lemma 4.4 and Theorem 1.2(b), we infer that, for all k, the automaton is not $(k+1)$-testable, and hence that it is not locally testable, contrary to the hypothesis in our lemma. □

DEFINITION 4.6. *Where G is a directed graph, $EP(G)$ is the table telling, for each ordered pair of nodes (v, v') of G, whether or not there exists a path in G from v to v'. If in addition G is acyclic then we define $LP(G)$ as the table telling, for each (v, v'), whether there exists a path from v to v' and, if so, the length of the longest such path.*

LEMMA 4.7. *Given a directed graph G, $EP(G)$ can be constructed in time $O(|G|^2)$.*

Proof. For a fixed vertex v of G, the entries for (v, v') where v' ranges over all vertices of G can all be entered in time $O(|G|)$ by a depth-first search for all the descendants of v. The algorithm for our lemma simply repeats this step for each vertex of G. □

LEMMA 4.8. *Given a directed graph G, the graph G^{ac} can be constructed in time $O(|G|)$.*

Proof. Let α be the set of nodes v of G that have no loops. The procedure to construct G^{ac} begins by constructing the set α. It does this by first executing an algorithm due to Tarjan that lists all SCCs of G in time $O(|G|)$ [1], [11]. Now each node v is in α if and only if $\{v\}$ is a singleton SCC and, for all $a \in \Sigma$, $\delta(v, a) \neq v$. Thus α is constructed in time $O(|G|)$. Finally $G^{ac} = G^{V-\alpha}$ is constructed from G and α in time $O(|G|)$. □

LEMMA 4.9. *If G is an acyclic directed graph then the table $LP(G)$ can be constructed in time $O(|G|^2)$ from G.*

Proof. The first step is to construct for each node v of G a list of nodes to which there is an edge from v, and a list of nodes from which there is an edge to v (if such lists are not already available). The time for this step is $O(|G|)$.

Let m be the number of edges of G and n the number of nodes. The second step is to give the nodes of G the names $v_1, \ldots, v_n$ in such a way that $(v_i, v_j) \in E$ implies $i < j$. The technique to accomplish this is well known (e.g., [7, pp. 259–265]); the time is $O(m+n) = O(G)$.

The third step is to fill in the entries of our table $LP(G)$ in which v_1 is the origin vector. The entries for $(v_1, v_2), (v_1, v_3), \ldots$, are computed in order. Assuming that the entries for $(v_1, v_2), \ldots, (v_1, v_j)$ have all been filled in, we consider each of the nodes that are edge-connected to v_{j+1}. We find the maximum path length from v_1 to one of these vertices; the maximum path length from v_1 to v_{j+1} equals one plus that value. (If there is no path from v_1 to any of the vertices that are edge-connected to v_{j+1} then, of course, there is no path from v_1 to v_{j+1}.) The time to compute the table entries for $(v_1, v_2), (v_1, v_3), \ldots, (v_1, v_n)$ is $O(m+n)$, since each edge is involved only once during this entire step.

This step is repeated for all entries whose origin node is v_2, with the same time analysis, then for v_3, v_4, etc. Thus the time for the entire algorithm ($n+2$ steps) is $O((n+2)(m+n))$, and is therefore $O(|G|^2)$. □

LEMMA 4.10. *There is an algorithm that, given G_2 and H_5, computes in time $O(|M|^{10})$ the table telling, for all nodes N and N' of H_5, the length of the longest* QCP *(if any) from N to N'.*

Proof. The proof is by Lemmas 4.5, 4.7, 4.8, and 4.9 and Definition 4.2 (QCP). □

THEOREM 4.11. *There is an algorithm that, given a locally testable* DFA *M, computes the smallest k satisfying condition* (ii) *of Theorem* 1.2(b) *in time $O(|M|^{10})$.*

Proof. The graphs G_2 and H_5^{ac} can be constructed in time $O(|M|^5)$ by Lemmas 4.3 and 4.8. The table of Lemma 4.10 can be constructed in time $O(|M|^{10})$. By Lemma 4.4, this table enables us to determine the smallest k satisfying condition (ii). □

We have shown, in effect, how to deal with condition (ii) of Theorem 1.2(b) in our computation of the order of a locally testable automaton. In the next section we focus on condition (i), which is the bigger obstacle.

5. Accommodating condition (i). Theorems 3.9 and 4.11 imply that, given a locally testable M, computing the smallest k satisfying condition (i) of Theorem 1.2(b) is NP-hard, so we must resort to approximation techniques for this subproblem of finding the order.

Our idea for the approximation begins with Theorem 2.2, which permits us to revise condition (i) of the condition for $(k+1)$-testability to an alternative condition on k.

Condition (i′). For all $q \in Q$, $u, v \in \Sigma^*$ and $h \geq 0$, $h|uv| + |u| = k$ implies $\delta(q, (uv)^{h+1}u) = \delta(q, (uv)^{h+2}u)$.

Conditions (i) and (i′) are related by the equations $x = (uv)^h u$, $y = vu$ and $z = uv$, justified by Theorem 2.2. Theorem 1.2 is valid when condition (i) is replaced by condition (i′).

Ideally, we should search for the largest k not satisfying condition (i′), considering each triple (q, u, v) to find the largest h such that $\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+2}u)$. Lemma 5.2 below will assure us that h exists, and also that it is the unique value of h for which the following relations hold:

$$\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+2}u) = \delta(q, (uv)^{h+3}u).$$

Having found h, it is the length of $x = (uv)^h u$ that is significant. In fact, we should search for the length of the longest such x, over all triples (q, u, v). Our result in §3 indicates that this search cannot be done in polynomial time. However, if we choose a fixed positive $\overline{h}$ and restrict our investigation of u's, v's, and h's to cases where $h \leq \overline{h}$, we do get a polynomial-time algorithm, with the degree of the polynomial depending on $\overline{h}$. It will turn out that by so doing we can compute the smallest k satisfying condition (i′) approximately. The degree of approximation will depend only on $\overline{h}$ and will be valid for all locally testable finite automata M, however large.

Our algorithm will assume a specification of the desired accuracy of the approximation, represented in the familiar way by a positive real number ϵ. The value of $\overline{h}$ depends on ϵ, but for the moment we ignore ϵ and assume that $\overline{h}$ is given. It is the purpose of this section to present this procedure in detail and to justify it. We begin with two rudimentary lemmas and a corollary.

LEMMA 5.1. *For any M, let $p \in Q$, $u, v \in \Sigma^*$, $|uv| \geq 1$, $k \geq 1$, and $h \geq i \geq 0$. Then, for any $u_1, u_2, v_1, v_2 \in \Sigma^*$, such that $u = u_1u_2$ and $v = v_1v_2$,*

(a) *$\delta(p, (uv)^i u_1) = \delta(p, (uv)^{i+k}u_1)$ implies $\delta(p, (uv)^h u) = \delta(p, (uv)^{h+k}u)$, and*

(b) *$\delta(p, (uv)^i uv_1) = \delta(p, (uv)^{i+k}uv_1)$ implies $\delta(p, (uv)^{h+1}u) = \delta(p, (uv)^{h+1+k}u)$.*

Proof. (a) Note that $(uv)^i u_1 = u_1(u_2vu_1)^i$. We can therefore rewrite the hypothesis in (a) as $\delta(p, u_1(u_2vu_1)^i) = \delta(p, u_1(u_2vu_1)^{i+k})$. Where $q_1 = \delta(p, u_1(u_2vu_1)^i)$, we see that $\delta(q_1, (u_2vu_1)^k) = q_1$, i.e., q_1 has a $(u_2vu_1)^k$ loop of length $k|uv|$. Notice that

$$(uv)^h u = (uv)^i u(vu)^{h-i} = (uv)^i u_1u_2(vu)^{h-i} = u_1(u_2vu_1)^i(u_2vu_1)^{h-i}u_2.$$

Thus, the node $q_2 = \delta(p, (uv)^h u) = \delta(q_1, (u_2vu_1)^{h-i}u_2)$ is also on the loop corresponding to the transition $\delta(q_1, (u_2vu_1)^k) = q_1$. So we have $\delta(q_2, (vu)^k) = q_2$. In other terms, $\delta(p, (uv)^h u(vu)^k) = \delta(p, (uv)^{h+k}u) = \delta(p, (uv)^h u)$, proving part (a). The proof of part (b) is similar. □

LEMMA 5.2. *Given $u, v \in \Sigma^*$, and $p \in Q$, let h be the smallest nonnegative integer such that, for some $k \geq 1$, $\delta(p, (uv)^h u) = \delta(p, (uv)^{h+k}u)$. If the smallest such k is greater than* 1 *then M is not locally testable.*

Proof. Let $q_1 = \delta(p, (uv)^h u)$. Then we have

$$\delta(p, (uv)^h u) = q_1 = \delta(p, (uv)^{h+k}u) = \delta(p, (uv)^h u(vu)^k) = \delta(q_1, (vu)^k),$$

i.e., q_1 has a $(vu)^k$ loop. If the smallest such $k > 1$, then $q_2 = \delta(q_1, vu) \neq q_1$. Clearly, $q_2 = \delta(q_2, (vu)^k)$, i.e., q_2 has a $(vu)^k$ loop. There are two distinct states q_1 and q_2 in an SCC that have a $(vu)^k$ loop. By Theorem 1.3(a) M is not locally testable. □

COROLLARY 5.3. *If M is locally testable, then, for any $q \in Q$, $u, v \in \Sigma^*$, there exists a nonnegative integer h such that*

(a) $\delta(q, (uv)^h u) = \delta(q, (uv)^{h+\beta} u)$ *for all* $\beta \geq 1$, *and*

(b) *for all* γ *and* γ' *such that* $\gamma < h$ *and* $\gamma \neq \gamma'$, $\delta(q, (uv)^{\gamma} u) \neq \delta(q, (uv)^{\gamma'} u)$.

Before embarking on the theoretical justification of our procedure, we consider an example. Assume $\overline{h} = 20$, $x = (uv)^{50}u$, and $\delta(q, (uv)^{51}u) \neq \delta(q, (uv)^{52}u)$. Here $h = 50$, and we must show that there is an $h' < 20$, and strings u' and v' such that $x' = (u'v')^{h'}u'$ fails to satisfy condition (i) and the length of x' will be almost that of $x = (uv)^{50}u$. We accomplish this by taking $u'v' = (uv)^s$, for some appropriate s, $u' = (uv)^i u$, for some $i \leq s - 1$, and $v' = v(uv)^{s-i-1}$. Then if s has been well chosen we shall be able to select $h' < 20$ and $i \leq s - 1$ so that $x' = (u'v')^{h'}u'$ can play the desired rôle. For this example, we see that if we take $s = 3$, $h' = 15$, and $i = 1$ (so that $u' = uvu$ and $v' = vuv$), we get $x' = (u'v')^{15}u' = (uv)^{46}u$. Furthermore, Corollary 5.3 implies that $\delta(q, (u'v')^{16}u') = \delta(q, (uv)^{49}u) \neq \delta(q, (u'v')^{17}u') = \delta(q, (uv)^{52}u)$, since $\delta(q, (uv)^{51}u) \neq \delta(q, (uv)^{52}u)$. Thus $\delta(q, (u'v')^{h'+1}u') \neq \delta(q, (u'v')^{h'+2}u')$, showing that $k = h'|u'v'| + |v'|$ does not satisfy condition (i′), and hence this k (which is $|x'|$) does not satisfy condition (i), as required. Of interest are the comparative lengths of x and x' as given by the fraction

$$\frac{|x| - |x'|}{|x'|} \leq \frac{4}{46}.$$

In general, suppose we have M, q, and $x = (uv)^h u$ such that $\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+2}u)$ but $h > \overline{h}$. In order to justify our algorithm, we prove the existence of integers $s \geq 2$, $i \leq s - 1$, and $h' \leq \overline{h}$, such that, for u' and v' determined as above, $x' = (u'v')^{h'}u'$ also satisfies the desired property and $|x'|$ is appropriately close to $|x|$. We postpone the consideration of what s must be until after the next lemma, and we assume for the moment that s is given.

LEMMA 5.4. *Where M is locally testable, suppose that* $2 \leq s \leq h$, $x = (uv)^h u$, $|uv| > 0$, *and* $\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+2}u)$, *for some* $q \in Q$. *Then, if we take* $h' = \lfloor (h+1)/s \rfloor - 1$ *and* $i = (h+1) \bmod s$, *we get*

$$\text{(a) } \delta(q, (u'v')^{h'+1}u') \neq \delta(q, (u'v')^{h'+2}u')$$

and

$$\text{(b) } \frac{|x|}{|x'|} \leq \frac{h}{h - (s-1)}$$

where, as above, $x' = (u'v')^{h'}u'$, $u' = (uv)^i u$, $v' = v(uv)^{s-i-1}$ *and hence* $u'v' = (uv)^s$.

Intuitively, Lemma 5.4 says that, given x and s, we can find an $x' = (u'v')^{h'}u'$, related to x as described in the paragraphs above, where $|x|/|x'| < h/(h - (s-1))$. The significance of this is that, if s had been chosen appropriately, h' would be no greater than $\overline{h}$, and $|x'|$ would be a good approximation to $|x|$. The lemmas that follow will make this statement precise and will justify it.

Proof of Lemma 5.4. (a) Note that $h' + 1$ is the quotient and i the remainder when $h + 1$ is divided by s. Thus $h + 1 = s(h' + 1) + i$. We get

$$\delta(q, (u'v')^{h'+2}u') = \delta(q, (uv)^{s(h'+2)}(uv)^i u) = \delta(q, (uv)^{s(h'+1)+i+s}u) = \delta(q, (uv)^{h+1+s}u),$$

and

$$\delta(q, (u'v')^{h'+1}u') = \delta(q, (uv)^{s(h'+1)+i}u) = \delta(q, (uv)^{h+1}u).$$

But, by Corollary 3.5 and $\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+2}u)$, it follows that $\delta(q, (uv)^{h+1}u) \neq \delta(q, (uv)^{h+1+s}u)$, which gives us part (a) of our lemma.

For part (b) note that

$$|x'| = h'|u'v'| + |u'| = sh'|uv| + i|uv| + |u| = (sh' + i)|uv| + |u|,$$

and $|x| = h|uv| + |u|$. For convenience, take $\alpha = |u|/|uv|$. Then, from $sh' + i = s(h' + 1) + i - s = h + 1 - s = h - (s - 1)$, we get

$$\frac{|x|}{|x'|} = \frac{h + \alpha}{sh' + i + \alpha} = \frac{h + \alpha}{h - (s - 1) + \alpha} \leq \frac{h}{h - (s - 1)},$$

since $\alpha \geq 0$ and $h > h - (s - 1) > 0$. □

We now take on the task of chosing a value for s. One consideration is that s should be as small as possible to make the ratio $h/(h - (s - 1))$ of Lemma 5.4 as small as possible. However, there is a restraint: we must make sure that $h' \leq \bar{h}$. So we must take s as the smallest positive integer such that $h' = \lfloor (h + 1)/s \rfloor - 1 \leq \bar{h}$. We need a lemma from elementary number theory.

LEMMA 5.5. *If a and b are positive integers and $a > b$, then thê smallest positive integer c such that $\lfloor a/c \rfloor \leq b$ equals $\lfloor a/(b + 1) \rfloor + 1$.*

Proof. Consider the set

$$\{x \mid \lfloor a/x \rfloor \leq b\} = \{x \mid a/x < b + 1\} = \{x \mid x > a/(b + 1)\}.$$

The smallest integer in this set is $\lfloor a/(b + 1) \rfloor + 1$. □

Applying Lemma 5.5 to the discussion immediately preceding, we see that we should take $s = \lfloor (h + 1)/(\bar{h} + 2) \rfloor + 1$. Lemma 5.5 tells us that this value of s gives us the optimal approximation. The remaining lemmas, which are proved without reference to Lemma 5.5, tell us how good the approximation is.

LEMMA 5.6. *Suppose it is given that $h > \bar{h} \geq 1$ and $x = (uv)^h u$, for a given M. Taking $s = \lfloor (h + 1)/(\bar{h} + 2) \rfloor + 1$ and h', u', v', and x' as in Lemma 5.4, we get $h' \leq \bar{h}$ and $|x|/|x'| \leq (\bar{h} + 1)/\bar{h}$.*

Proof. First note that $2 \leq s \leq h$, which shows that Lemma 5.4 is applicable. We are assured by the informal discussion since Lemma 5.4 that $h' \leq \bar{h}$. By Lemma 5.4, $|x|/|x'| \leq h/(h - (s - 1))$. Since $s - 1 = \lfloor (h + 1)/(\bar{h} + 2) \rfloor \geq 1$, if we take $r = (h + 1) - (s - 1)(\bar{h} + 2)$ then we have $0 \leq r \leq \bar{h} + 1$ and $h = (s - 1)(\bar{h} + 2) + r - 1$. (Note that $s - 1$ and r are quotient and remainder when $h + 1$ is divided by $\bar{h} + 2$.) Thus

$$\frac{h}{h - (s - 1)} = \frac{(s - 1)(\bar{h} + 2) + r - 1}{(s - 1)(\bar{h} + 1) + r - 1} = \frac{\bar{h} + 2 + (r - 1)/(s - 1)}{\bar{h} + 1 + (r - 1)/(s - 1)} \leq \frac{\bar{h} + 1}{\bar{h}},$$

since $1 + (r - 1)/(s - 1) \geq 1 - 1/(s - 1) \geq 0$. □

The significance of Lemma 5.6 is that, having picked $\bar{h}$, we can be sure that, for any locally testable automaton M, however large, and for any $u, v \in \Sigma^*$ and h, however large, the following will hold: for $x = (uv)^h u$, there exist u', v', and $h' \leq \bar{h}$ such that we can use $x' = (u'v')^{h'} u'$ instead of x to compute the largest k not satisfying condition (i′), and our result will be off by a quantity that is a function of $\bar{h}$ only. This implies that if we compute for any M the largest k not satisfying condition (i′), ignoring strings of the form $(uv)^h u$ where $h > \bar{h}$, the result will be off by at most a quantify that is a function of $\bar{h}$ only. Furthermore, the value of this function can be made as small as we please provided we are willing to pay the penalty of a sufficiently large $\bar{h}$.

Under the assumption that $\overline{h}$ is fixed, we consider the problem of finding, for any given M, the largest k not satisfying the following condition.

Condition (i″). For all $q \in Q$, $u, v \in \Sigma^*$ and $h \leq \overline{h}$, $h|uv| + |u| = k$ implies $\delta(q, (uv)^{h+1}u) = \delta(q, (uv)^{h+2}u)$.

Note that condition (i″) is like condition (i′) except for the upper limit on h. Unlike condition (i′), condition (i″) is not equivalent to condition (i) of Theorem 1.2(b). Although it cannot be used to find the order of a locally testable automaton, it will be used to find an approximate order.

DEFINITION 5.7 (n-CP). *For $n \geq 2$, let $P = N_0N_1 \ldots N_m$, $m \geq 0$, be a path on I_n, where $N_j = (p_{1,j}, p_{2,j}, \ldots, p_{n,j})$, $p_{i,j} \in Q$ for $0 \leq j \leq m$, and $1 \leq i \leq n$. We say path P is an* n-CP (n-fold coherent path) *if and only if I_{n-1} has a path, called a* connecting path, *of length ≥ 0 from $(p_{1,m}, p_{2,m}, \ldots, p_{n-1,m})$ to $(p_{2,0}, p_{3,0}, \ldots, p_{n,0})$.*

Note that QCPs (Definition 4.2) and 5-CPs are different. Also note that the length m of an n-CP and the length of the connecting path cannot both be 0, since that would force $p_{1,0} = p_{1,m} = p_{2,0}$, and N_0 would not be a node of I_n. The significance of n-CPs will be demonstrated in Lemmas 5.8 and 5.9.

LEMMA 5.8. *Let M be a locally testable automaton. For $p_{1,0} \in Q$ and $h \geq 0$, if $\delta(p_{1,0}, (uv)^{h+1}u) \neq \delta(p_{1,0}, (uv)^{h+2}u)$ then there is an $(h+3)$-*CP *spanned by u whose connecting path is spanned by v.*

Proof. Let $m = |u|$. Take $N_0 = (p_{1,0}, p_{2,0}, \ldots, p_{h+3,0})$ and $N_j = (p_{1,j}, \ldots, p_{h+3,j})$, where, for each i and j, $p_{i,j} = \delta(p_{i,0}, u_j)$, where u_j is the prefix of u of length j, and $p_{i+1,0} = \delta(p_{i,m}, v)$. Also, take $L_0 = (p_{1,m}, \ldots, p_{h+2,m}) = (q_{1,0}, \ldots, q_{h+2,0})$ and, for each j such that $1 \leq j \leq |v|$, $L_j = (q_{1,j}, \ldots, q_{h+2,j})$, where, for $1 \leq i \leq h+2$, $q_{i,j} = \delta(q_{i,0}, v_j)$, v_j being the prefix of v of length j. Thus $L_{|v|} = (q_{1,|v|}, \ldots, q_{h+2,|v|}) = (p_{2,0}, \ldots, p_{h+3,0})$. We proceed to demonstrate that $N_0N_1 \ldots N_m$ is an $(h+3)$-CP spanned by u. Actually, it is quite clear that this is so if $N_0, \ldots, N_m$ are all nodes of I_{h+3} and $L_0, \ldots, L_{|v|}$ are all nodes of I_{h+2}, which we now prove by contradiction.

Accordingly, assume $N_j = (p_{1,j}, \ldots, p_{h+3,j})$ is not a node of I_{h+3}. Then $p_{i,j} = p_{i+k,j}$ for some $i \geq 1$ and $k \geq 1$, where $i+k \leq h+3$. Assume, without loss of generality, that k is the smallest integer satisfying this equation for any i and j. Note that $p_{i,j} = \delta(p_{1,0}, (uv)^{i-1}u_j)$, for all i and j.

Case I. $k = 1$. Then $\delta(p_{1,0}, (uv)^i u_j) = \delta(p_{1,0}, (uv)^{i+1}u_j)$. By Lemma 5.1, part (a), $\delta(p_{1,0}, (uv)^{h+1}u) = \delta(p_{1,0}, (uv)^{h+2}u)$, which contradicts the hypothesis of our lemma.

Case II. $k > 1$. Then $\delta(p_{1,0}, (uv)^{i-1}u_j) = p_{i,j} = p_{i+k,j} = \delta(p_{1,0}, (uv)^{i-1+k}u_j)$, but $p_{i,j} \neq p_{i+1,j} = \delta(p_{i,0}, (uv)^i u_j)$, which, by Lemma 5.2, implies that M is not locally testable, again contradicting the hypothesis of our lemma.

It remains to prove that the nodes $L_0, \ldots, L_{|v|}$ are in I_{h+2}. The proof is similar to the above proof for $N_0, \ldots, N_m$, using part (b) of Lemma 5.1 instead of part (a). □

The following lemma follows directly from the definition of n-CP.

LEMMA 5.9. *For $h \geq 0$, if there is an $(h+3)$-CP $N_0 \ldots N_m$ spanned by a string u and with a connecting span v then $\delta(p_{1,0}, (uv)^{h+2}u) \neq \delta(p_{1,0}, (uv)^{h+1}u)$, where $p_{1,0}$ is the first coordinate of N_0.*

LEMMA 5.10. *If M is locally testable then* (a) *all $(h+3)$-CPs are in I^{ac}_{h+3}, for all $h \geq 0$, and* (b) *the connecting paths of all $(h+3)$-CPs are in I^{ac}_{h+2} for all $h \geq 1$.*

Proof. Let $N_0 \ldots N_m$ be an $(h+3)$-CP, $N_i = (p_{1,i}, \ldots, p_{h+3,i})$, $0 \leq i \leq m$. For (a), assume that N_j $(0 \leq j \leq m)$ is not in I^{ac}_{h+3}. Then N_j has a loop and we would have arbitrarily long $(h+3)$-CPs from N_0 to N_m (cf. the proof of Lemma 4.5). Thus, by Lemma 5.9, for arbitrarily long u and for some fixed v, $\delta(p_{1,0}, (uv)^{h+2}u) \neq \delta(p_{1,0}, (uv)^{h+1}u)$, where $p_{1,0}$ is the first coordinate of N_0. But this would imply, by Theorems 1.2(b) (i) and 2.2 (cf. our discussion of condition (i′) above), that, for all $k \geq 1$, M is not k-testable. Thus M would not be locally testable. This establishes part (a).

For (b), assume $h \geq 1$ and assume that the connecting path from $(p_{1,m}, \ldots, p_{h+2,m})$ to $(p_{2,0}, \ldots, p_{h+3,0})$ has a node which has a loop. Where u is the span of the $(h+3)$-CP, we see that the $(h+3)$-CP has arbitrarily long connecting string v. For each v, $\delta(p_{1,0}, (uv)^{h+2}u) \neq \delta(p1, 0, (uv)^{h+1}u)$. Since the v's are arbitrarily long and $h \geq 1$, again by Theorems 1.2(b)(i) and 2.1, M is not locally testable. □

Note that part (b) of Lemma 5.10 does not apply to the case $h = 0$, viz., the connecting paths of 3-CP's need not be in I_2^{ac}. For in the case $h = 0$ we get $\delta(p_{1,0}, (uv)^1u) \neq \delta(p_{1,0}, (uv)^2u)$ from which we take $x = (uv)^0u = u$ and say that M is not $(|x|+1)$-testable. If there is a loop on a node in the connecting path then we do get arbitrarily long v's, but this does not give us arbitrarily long x's.

6. The approximation algorithm and its analysis. Our algorithm begins with a given M known to be locally testable. The largest k_2 not satisfying condition (ii) of Theorem 1.2(b) is determined without regard to ϵ, as discussed in §4. Then the algorithm computes $\bar{h}$ from ϵ (in a manner to be described), constructs I_2, and for each j, $3 \leq j \leq \bar{h}+3$, constructs the graph I_j^{ac} and the table $LP(I_j^{ac})$, using Lemmas 4.3, 4.8, and 4.9. (In some cases the algorithm will not need to go as far as $j = \bar{h}+3$, as we shall see.)

For each $h \leq \bar{h}$ and for each pair of nodes $N = (p_1, \ldots, p_{h+3})$ and $N' = (p'_1, \ldots, p'_{h+3})$ of I_{h+3}^{ac}, we define $\Delta_{h+3}(N, N')$ as follows. If there is no path from N to N' in I_{h+3}^{ac} or if there is no path in I_{h+2}^{ac} from $(p'_1, \ldots, p'_{h+2})$ to $(p_2, \ldots, p_{h+3})$ then $\Delta_{h+3}(N, N') = 0$. Otherwise, let $d_1 = d_1(N, N')$ be the maximum length of a path in I_{h+3}^{ac} from N to N' and $d_2 = d_2(N, N')$ be the maximum length of a path in I_{h+2}^{ac} from $(p'_1, \ldots, p'_{h+2})$ to $(p_2, \ldots, p_{h+3})$, and take $\Delta_{h+3}(N, N') = h(d_1 + d_2) + d_1$.

Note that for $h = 0$, d_2 has no material affect on $\Delta_{h+3}(N, N')$, as we discussed at the end of the preceding section. Hence in our algorithm we do not compute d_2 in the case $h = 0$. However, the algorithm must still verify that a connecting path in I_2 exists.

We take $\Delta(h+3)$ to be the maximum value of $\Delta_{h+3}(N, N')$ over all pairs of nodes (N, N') of I_{h+3}^{ac}. Finally, we take

$$k_1 = \mathrm{MAX}_{0 \leq h \leq \bar{h}}(\Delta(h+3)).$$

The proofs of the following two lemmas are straightforward from the definitions of $\Delta_{h+3}(N, N')$ and $(h+3)$-CP.

LEMMA 6.1. *For any nodes N and N' of I_{h+3}^{ac}, if $\Delta_{h+3}(N, N') = 0$, then there is no $(h+3)$-CP from N to N'. But if $\Delta_{h+3}(N, N') > 0$, then there exist an $(h+3)$-CP P from N to N', and strings $u, v \in \Sigma^*$ such that*

(a) *u is a span of P and, for $h > 0$, v is a span of a connecting path of P,*

(b) *$|u| = d_1(N, N')$, $|v| = d_2(N, N')$, and, hence, $|(uv)^h u| = \Delta_{h+3}(N, N')$, for some $h > 0$, and*

(c) *for any $(h+3)$-CP from N to N' with a span u' and a connecting path with a span v', it satisfies $|(u'v')^h u'| \leq |(uv)^h u|$.*

LEMMA 6.2. *For a given $\bar{h}$, the largest k not satisfying condition* (i″) *of* §5 *is the k_1 determined above.*

We now determine how to compute $\bar{h}$ from ϵ. Lemma 5.6 tells us that, if $x = (uv)^h u$ with $h > \bar{h}$, we can get $x' = (u'v')^{h'}u'$ with $h' \leq \bar{h}$ and $|x|/|x'| \leq (\bar{h}+1)/\bar{h}$. We want $(|x| - |x'|)/|x'|$ to be $\leq \epsilon$, and hence we want $|x|/|x'|$ to be $\leq 1+\epsilon$. We can accomplish this by taking $\bar{h}$ so that $(\bar{h}+1)/\bar{h} \leq 1+\epsilon$, or $\bar{h} \geq 1/\epsilon$. So we take $\bar{h} = \lceil 1/\epsilon \rceil$.

The value of k_2 that is computed is exact but (except for the circumstances noted below) the value of k_1 that is computed is only approximately the value needed to compute the order.

These values could be used to compute an approximation to the order k_{or} of M. But the approximation, if not exact, is less than k_{or}, since k_1 is generally less than the true value needed for condition (i) of Theorem 1.2(b). As we explained at the beginning of §4, we prefer to have the approximate order greater than k_{or} rather than less, so a modification must be made.

Recall that k_1 is the length of the longest $x' = (u'v')^{h'}u'$, for some u', v' and $h' \leq \bar{h}$, such that k_1 does not satisfy condition (i) of Theorem 1.2(b). Let $x = (uv)^h u$ be a longest x, with no limit on h, such that x does not satisfy that condition; let $c = |x|$. We know that $k_1 \leq c$ and $(c - k_1)/k_1 \leq \epsilon$. The next lemma enables us to find an integer k_1' such that $k_1' \geq c$ and $(k_1' - c)/c \leq \epsilon$.

LEMMA 6.3. *Let k_1 and c be as described above and $k_1' = k_1(1 + \epsilon)$; then $k_1' \geq c$ and $(k_1' - c)/c \leq \epsilon$.*

Proof. Where x and x' are as described above, Lemma 5.6 tells us that

$$\frac{c}{k_1} = \frac{|x|}{|x'|} \leq \frac{\bar{h}+1}{\bar{h}} = \frac{\lceil 1/\epsilon \rceil + 1}{\lceil 1/\epsilon \rceil} \leq \frac{1/\epsilon + 1}{1/\epsilon} = 1 + \epsilon.$$

Hence $k_1' = k_1(1 + \epsilon) \geq c$. Also, since $k_1 \leq c$, it follows that

$$\frac{k_1' - c}{c} \leq \frac{k_1' - k_1}{k_1} = \frac{k_1'}{k_1} - 1 = 1 + \epsilon - 1 = \epsilon. \qquad \square$$

There are two circumstances in which execution of the algorithm permits it to assert that the exact order has been computed. The first is the discovery of an $h \leq \bar{h} + 3$ such that there are no h-CPs, revealed in the result that $\Delta(h) = 0$. From this fact it follows that there are no h'-CPs for any $h' > h$. (From the definition of n-CP, it is easy to see that $h' > h$ implies that every h'-CP can be converted into an h-CP.) Thus execution in this circumstance reveals with certainty that all the n-CPs needed to compute the exact order have been examined.

The second such circumstance is that in which k_1, after being multiplied by $1 + \epsilon$, is no greater than k_2. Since k_2 has been computed with absolute precision, in this circumstance also it is known with certainty that the exact order has been computed.

The approximation algorithm is given in schematic form in Fig. 8. The correctness of the algorithm will follow from our work above once we prove the following lemma.

LEMMA 6.4. *The assertions $k_1 \geq c$ and $(k_1 - c)/c \leq \epsilon$ after line* 12 *of the algorithm are justified.*

Proof.

Case I. $i < \bar{h}+3$ at line 12. Then $k_1 = c$ is true upon leaving the loop under the condition that $\Delta(i + 1) = 0$, as explained above. Our lemma is therefore true in this case.

Case II. $i = \bar{h} + 3$ at line 12. Let k_{10} and k_{11}, respectively, be the values of k_1 before and after line 12 has been executed. Then $k_{11} = \lfloor k_{10}(1 + \epsilon) \rfloor$. We know $(c - k_{10})/k_{10} \leq \epsilon$ or, equivalently, $c \leq k_{10}(1 + \epsilon)$. From the fact that c is an integer it follows that $k_{11} \geq c$. Lemma 6.3 tell us that $(k_{11} - c)/c \leq \epsilon$. $\square$

In analyzing this algorithm, we note that it is dominated by the time it takes to construct the various graphs and their longest-path tables. Put $|M| = m$. The time to compute k_2 is $O(m^{10})$ by Theorem 4.11 (the k of Theorem 4.11 is one more than k_2).

To analyze the computation of k_1 we first note that each of the constructions in the while-loop of line 10, for a given value of i, is done in time

$$O(|G_{i+1}|^2) = O((m^{i+1})^2) = O(m^{2i+2})$$

by Lemmas 4.3, 4.8, and 4.9. We can ignore what is outside the loop; thus the time to compute k_1 is

$$O(\sum_{i=3}^{\bar{h}+2} m^{2i+2}) = O(m^{2\bar{h}+6}).$$

Thus the time for the execution of the algorithm is $O(|M|^{\mathrm{MAX}(10,6+2\bar{h})})$, where $\bar{h} = \lceil 1/\epsilon \rceil$. We see that the algorithm is less costly to operate the larger the value of ϵ, but increasing ϵ to values greater than $1/2$ is to no avail. Taking $\epsilon = 1/2$ allows the algorithm to operate in time $O(|M|^{10})$. In this case $\bar{h} = 2$ and n-tuple graphs are constructed only as far as G_5, which are precisely those used to compute k_2 in the algorithm, and which therefore would be needed even if ϵ were increased.

```
Procedure ApproxOrder($M$, $\epsilon$);

begin
    (// We first compute $k_2$. //)
1.  Construct $G_1$ from $M$; $G_2$ from $G_1$;
2.  Construct $EP(G_2)$; Construct $G_3$ from $G_2$ and $G_1$;
3.  Construct $G_5$ from $G_3$ and $G_2$; $H_5$ from $G_5$; $H_5^{ac}$ from $H_5$; Construct $LP(H_5^{ac})$;
4.  Find the length $k_2$ of the longest QCP from $LP(H_5^{ac})$ and $EP(G_2)$;

    (// We now compute $k_1$. //)
5.  $\bar{h} := \lceil 1/\epsilon \rceil$;
6.  Construct $I_2$ from $G_2$; Construct $EP(I_2)$;
7.  Construct $I_3$ from $G_3$; $I_3^{ac}$ from $I_3$; Construct $LP(I_3^{ac})$;
8.  Compute $\Delta(3)$ from $LP(I_3^{ac})$ and $EP(I_2)$;
9.  $i := 3$; exit := false;

10. While (($i \leq \bar{h} + 2$) and (not exit)) do
        begin
            Construct $I_{i+1}$ from $I_i$ and $G_1$; $I_{i+1}^{ac}$ from $I_{i+1}$; Construct $LP(I_{i+1}^{ac})$;
            Compute $\Delta(i + 1)$ from $LP(I_{i+1}^{ac})$ and $LP(I_i^{ac})$;
            if $\Delta(i + 1) = 0$ then exit := true else $i := i + 1$
        end;

11. $k_1 := MAX_{3 \leq j \leq i}(\Delta(j))$;
12. if $i = \bar{h} + 3$ then $k_1 := \lfloor k_1(1 + \epsilon) \rfloor$;
    (// At this point, $k_1 \geq c$ and $(k_1 - c)/c \leq \epsilon$, where $c$ is the largest integer
        not satisfying condition (i). //)

    (// Finally the order, precise or approximate, is computed. //)
13. $k := MAX(k_1, k_2) + 2$;
14. if ($i < \bar{h} + 3$ or $k_2 \geq k_1$)
        then return ('We found the exact order:' $k$)
        else return ('We found an approximate order:' $k$)
end.
```

FIG. 8. *The ϵ-approximation algorithm.*

There is a question as to whether the complexity bound $O(|M|)^{\mathrm{MAX}(10,6+2\bar{h})})$ for the algorithm can be decreased. If one could rewrite the algorithm so that the maximum lengths of the i-CPs and QCPs could be computed in time linear in the graph involved, then it might be possible to get the time complexity of the algorithm down to $O(|M|^{\mathrm{MAX}(5,3+\bar{h})})$. As it is, the construction of the tables $LP(H_5^{ac})$ and $LP(I_i^{ac})$ requires time on the order of the square of the size of the graphs involved.

7. Conclusions. We have showed that computing the order k of a locally testable deterministic automaton is NP-hard and have presented a polynomial-time algorithm for ϵ-approximation. These results answer the open questions raised in [2].

Our approximation algorithm is a polynomial-time approximation scheme since, for any constant $\epsilon > 0$, it computes an approximate order $\hat{k}$ such that $\hat{k} \geq k$ and $(\hat{k} - k)/k \leq \epsilon$ in time polynomial in the size of the automaton. However, since the time complexity is exponential in $1/\epsilon$, it is not a fully polynomial-time approximation scheme in the sense of Chapter 12 of [5].

We are interested in knowing whether there is a practical method of determining, for certain small values of k, if a given automaton is k-testable. For example, could the easy algorithm for the 1-testability problem established by Theorem 2.1 be extended to 2-testability, 3-testability, etc., without an excessive increase in complexity? This question is a challenge for future research.

Theorem 2.8 shows an upper bound $2n^2 + 1$ on the order of a locally testable deterministic automaton, where n is the number of states of the automaton. In [6] we conjectured that the bound is $O(n^2)$ and, for automata whose alphabet size is two, $O(n^{1.5})$. Thus Theorem 2.8 verfies part of our conjecture. As we discussed in §2, we still think that the other part of our conjecture is true.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Uullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] J. Brzozowski and I. Simon, *Characterizations of locally testable events*, Discrete Math., 4 (1973), pp. 243–271.

[3] M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.

[4] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, MA, 1979.

[5] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1984.

[6] S. M. Kim, R. McNaughton, and R. McCloskey, *A polynomial time algorithm for the local testability problem of deterministic finite automata*, IEEE Trans. Comput., 40 (1991), pp. 1087–1093.

[7] D. Knuth, *The Art of Computer Programming Vol. 1*, Addison-Wesley, Reading, MA, 1968.

[8] M. Minsky and S. Papert, *Perceptrons*, M.I.T. Press, Cambridge, MA, 1969.

[9] R. McNaughton and S. Papert, *Counter-free Automata*, M.I.T. Press, Cambridge, MA, 1971.

[10] R. McNaughton, *Algebraic decision procedures for local testability*, Math. Systems Theory, 8 (1974), pp. 60–76.

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

[12] Y. Zalcstein, *Locally testable languages*, J. Comput. System Sci., 6 (1972), pp. 151–167.

SIAM J. COMPUT.
Vol.23, No.6, pp. 1216–1224, December 1994

A POLYNOMIAL-TIME ALGORITHM FOR THE PERFECT PHYLOGENY PROBLEM WHEN THE NUMBER OF CHARACTER STATES IS FIXED*

RICHA AGARWALA† AND DAVID FERNÁNDEZ-BACA‡

Abstract. This paper presents a polynomial-time algorithm for determining whether a set of species, described by the characters they exhibit, has a perfect phylogeny, assuming the maximum number of possible states for a character is fixed. This solves a longstanding open problem. This result should be contrasted with the proof by Steel [*J. Classification*, 9(1992), pp. 91–116] and Bodlaender, Fellows, and Warnow [*Proceedings of the* 19*th International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, 1992, pp. 273–283] that the perfect phylogeny problem is NP complete in general.

Key words. algorithms, character compatibility, evolution, perfect phylogeny

AMS subject classifications. 68Q25, 68R, 92A12, 92-08

1. Introduction. A fundamental problem in biology is that of inferring the evolutionary history of a set of species, each of which is specified by the set of *traits* or *characters* that it exhibits [9], [11], [17], [18]. Information about evolutionary history can be conveniently represented by an evolutionary or *phylogenetic* tree, often referred to simply as a *phylogeny*. In one of the standard models, the problem can be expressed mathematically as follows. Let $\mathcal{C} = \{1, \ldots, m\}$ be the *character set*, and for every $c \in \mathcal{C}$, let $\mathcal{A}_c = \{1, \ldots, r_c\}$ be the set of allowable *states* for character c. We write r to denote $\max_{c \in \mathcal{C}} r_c$. A *species* $\boldsymbol{s}$ is a vector $(s_1, \ldots, s_m)$ such that $\boldsymbol{s} \in \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$; s_c is referred to as the *state of character c for* $\boldsymbol{s}$, or the *state of* $\boldsymbol{s}$ *on character c*. We assume that if $i \in \mathcal{A}_c$, then there exists a species $\boldsymbol{s} \in \mathcal{S}$ such that $s_c = i$. The *perfect phylogeny problem* is to determine whether a given set of n distinct species $\mathcal{S}$ has a tree T with the following properties:

(C1) $\mathcal{S} \subseteq V(T) \subseteq \mathcal{A}_1 \times \cdots \times \mathcal{A}_m$,
(C2) Every leaf in T is in $\mathcal{S}$,
(C3) For every $c \in \mathcal{C}$ and every $j \in \mathcal{A}_c$, the set of all $\boldsymbol{u} \in V(T)$ such that $u_c = j$ induce a subtree of T.

The tree T, if it exists, is called a *perfect phylogeny* for $\mathcal{S}$ and the set of characters $\mathcal{C}$ is said to be *compatible*. We should point out that in the biology literature, the perfect phylogeny problem is more commonly known as the *character compatibility problem* [6]. In this context, one is frequently interested in computing a maximal set of compatible characters since in practice character sets tend to be incompatible. Note also that instances of the phylogeny problem are often expressed in matrix form, by giving the set of species $\mathcal{S}$ as an $n \times m$ matrix M whose rows are the species in $\mathcal{S}$ [5], [14].

The perfect phylogeny problem was shown to be NP complete by Bodlaender, Fellows, and Warnow [2] and, independently, by Steel [20]. This fact suggests at least two lines of attack: one is to restrict m, the number of characters; the other is to restrict r. Pursuing the first approach, McMorris, Warnow, and Wimer have shown that the perfect phylogeny problem is solvable in $O(n^{m+1})$ time [19], which is polynomial for every fixed m; linear time algorithms have been found for $m = 3$ [3], [15]. In this paper, we pursue the second approach. Gusfield [14] gave an $O(nm)$ algorithm for $r = 2$, the *binary* character case; the procedure

*Received by the editors February 22, 1993; accepted for publication (in revised form) July 15, 1993.

†Department of Computer Science, Iowa State University, Ames, Iowa 50011 (agarwala@iastate.edu). Supported in part by a College of Liberal Arts and Sciences research assistantship, Iowa State University.

‡Department of Computer Science, Iowa State University, Ames, Iowa 50011 (fernande@cs.iastate.edu). Supported in part by the National Science Foundation under grants CCR-8909626 and CCR-9211262. This author acknowledges the support of The Center for Discrete Mathematics and Theoretical Computer Science, at Rutgers University, where much of this work was carried out.

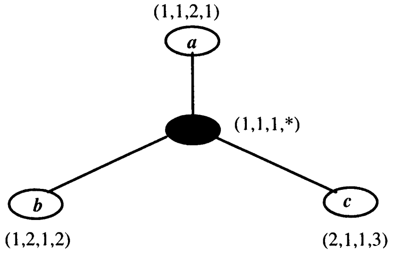


FIG. 1. *Forced and unforced states.*

is optimal if the input is given as a matrix of zeros and ones. In [1] it is shown that, under a suitable representation of the input, the run time can be reduced to $O(C)$ where C is the number of ones in the matrix. This reference also presents efficient algorithms for dynamic insertion and deletion of species and characters. All of the above results for the binary case are based on an elegant and well-known characterization of the set of "yes" instances [8]. Dress and Steel [7] devised an $O(nm^2)$ algorithm for $r \leq 3$. Kannan and Warnow [16] gave an $O(n^2m)$ algorithm for $r \leq 4$ (*quaternary* characters) and conjectured the existence of an $O(r^{r-2}n^2m)$ algorithm, which is polynomial for every fixed r. Here we prove the existence of a polynomial-time algorithm for any fixed r by giving an $O(2^{3r}(nm^3 + m^4))$ algorithm for the perfect phylogeny problem.

Instances of the perfect phylogeny problem where there is a known upper bound on the number of states per character are of interest to biologists. For instance, quaternary characters arise when using DNA to describe species; each possible state of a character corresponds to a nucleotide, A, G, C, or T. Other instances of interest are those where $r = 20$, which occur when species are described by protein sequences. These can be viewed as strings from a 20-letter alphabet, with each letter corresponding to one amino acid. While admittedly the running time of our algorithm grows quickly with r, it is a worst-case estimate which in practice may not pose too large a problem for current computational technology. Furthermore, the dependency on r can be reduced by a factor of 2^r using recent ideas due to Kannan and Warnow (see §6).

2. Preliminaries.

We now introduce some definitions and prove certain preliminary results.

DEFINITION 2.1. *Suppose T is a perfect phylogeny for $\mathcal{S}$ and let $\mathbf{p}$ be some vertex in T. We shall say that the state of $\mathbf{p}$ on character c is* forced *if $\mathbf{p}$ lies on the path between vertices $\mathbf{a}$ and $\mathbf{b}$ in $\mathcal{S}$ such that $a_c = b_c$. (Observe that if this is the case, in order to satisfy condition* (C3) *we must have $p_c = a_c = b_c$.)*

If the state of a character of a node is unforced, several assignments may be possible. In Fig. 1, for example, the state of the fourth character of the internal node is unforced and we could assign it a value of 1, 2, or 3.

LEMMA 2.2. *A set of species $\mathcal{S}$ has a perfect phylogeny if and only if every subset of $\mathcal{S}$ has one.*

Proof. The "if" part is trivial. For the "only if" part, let $\mathcal{S}'$ be any subset of $\mathcal{S}$ and let T be a perfect phylogeny of $\mathcal{S}$. Clearly, T satisfies (C1) and (C3) for $\mathcal{S}'$ but possibly not (C2). To obtain a perfect phylogeny for $\mathcal{S}'$, repeatedly delete from T any leaf that is not in $\mathcal{S}'$ until this operation is no longer possible. Since each deletion preserves properties (C1) and (C3) for $\mathcal{S}'$, the final tree will also satisfy (C2) for $\mathcal{S}'$. □

DEFINITION 2.3. *Suppose $G \subset \mathcal{S}$ and let $G' = \mathcal{S} - G$. $\mathcal{D}(G)$, the set of* distinguishing characters *of G, is the set of all $c \in \mathcal{C}$ such that for every $\mathbf{a} \in G$ and every $\mathbf{b} \in G'$, $a_c \neq b_c$. $\mathcal{M}(G)$, the set of* common characters, *is $\mathcal{C} - \mathcal{D}(G)$.*

Obviously, $\mathcal{D}(G) = \mathcal{D}(G')$ and $\mathcal{M}(G) = \mathcal{M}(G')$.

DEFINITION 2.4. *A pair (G, G') where $G \subset \mathcal{S}$ and $G' = \mathcal{S} - G$ is called a* split *if, for every character, the number of common character states between G and G' is at most one. A split (G, G') is a* c-split *if $\mathcal{D}(G) \neq \emptyset$. If (G, G') is a split (c-split), G and G' are called* clusters (c-clusters).

Although there can be up to $2^{n-1} - 1$ splits, the total number of c-splits is at most $(2^{r-1} - 1) \cdot m$. Observe that we can determine whether a partition (G, G') of $\mathcal{S}$ is a split in $O(nm)$ time. Note also that if G is a cluster but not a c-cluster, then $\mathcal{M}(G) = \mathcal{C}$.

DEFINITION 2.5. *Let (G, G') be a split. We say that (G, G') is of* type I *if there exists an $s \in G$ such that for all $c \in \mathcal{M}(G)$, s_c equals the unique common state between G and G' on character c and $|G - \{s\}|, |G'| \geq 1$. If (G, G') is of type* I, *we refer to s as a* connecting species. *If (G, G') is not of type* I, *we say that it is of type* II.

Checking whether a split (G_1, G_2) is of type I can be achieved in $O(nm)$ time as follows. First, compute $\mathcal{M}(G_1)$ and the common state between G_1 and G_2 on each $c \in \mathcal{M}(G_1)$. This can be done in $O(nm)$ time by considering characters one at a time. Now, it suffices to search for a species s such that for all $c \in \mathcal{M}(G_1)$, s_c equals the common state between G_1 and G_2. This takes $O(m)$ time per species, for a total of $O(nm)$ time.

If there is a type I c-split (G_1, G_2) where s is a connecting species, the problem can be divided into constructing perfect phylogenies T_1 and T_2 for $G_1 \cup \{s\}$ and $G_2 \cup \{s\}$. If one or both of the latter sets has no perfect phylogeny then, by Lemma 2.2, neither does $\mathcal{S}$. If both sets have perfect phylogenies, then a perfect phylogeny for $\mathcal{S}$ is obtained by identifying the nodes for s in T_1 and T_2. We now consider the case where all c-splits are of type II.

LEMMA 2.6. *If all c-splits are of type* II, *then, in every perfect phylogeny T of $\mathcal{S}$, every species $s \in \mathcal{S}$ is a leaf in T.*

Proof. Suppose there exists a species $s \in \mathcal{S}$ such that s is an internal node in some perfect phylogeny T of $\mathcal{S}$. Let T' be any connected component of $T - s$, let G' be the set of species in T', and let $G = \mathcal{S} - G'$. Clearly, (G, G') is a c-split with $|G'| \geq 1$ and $|\mathcal{S} - G'| \geq 2$. One can also readily verify that c-split (G, G') is of type I, with s as a connecting species. □

3. Subphylogenies.

In this section and the next we assume that $\mathcal{S}$ has no type I splits.

DEFINITION 3.1. *A* subphylogeny *T_G for a cluster G is a perfect phylogeny for G containing a node x such that for every $c \in \mathcal{M}(G)$, x_c equals the (unique) common state for character c between G and $\mathcal{S} - G$, and for every $c \in \mathcal{D}(G)$, x_c is the state of some species in G on character c. Node x is referred to as the* connection *of T_G.*

The next result implies that, in searching for a perfect phylogeny for $\mathcal{S}$, we can restrict our attention to perfect phylogenies constructed entirely from subphylogenies.

LEMMA 3.2. *$\mathcal{S}$ has a perfect phylogeny if and only if there exists a split (G_1, G_2) such that both G_1 and G_2 have subphylogenies.*

Proof. For the "if" part, let (G_1, G_2) be a split satisfying the requirements of the lemma and let T_1 and T_2 be subphylogenies for G_1 and G_2, respectively. Let x^1 and x^2 be the connections of T_1 and T_2. We can obtain a perfect phylogeny for $\mathcal{S}$ by taking T_1 and T_2 and connecting them as follows. If $\mathcal{D}(G_1) = \emptyset$, identify x^1 and x^2. Otherwise, add an edge (x^1, x^2). It is not hard to check that conditions (C1)–(C3) hold.

For the "only if" part, let T be a perfect phylogeny for $\mathcal{S}$ and let (u, v) be any edge in T. Without loss of generality, assume that every node in T that is not in $\mathcal{S}$ has a degree of at least three. Let T_1 and T_2 be the subtrees of $T - (u, v)$ containing u and v, respectively, and let $G_1 = \mathcal{S} \cap V(T_1)$ and $G_2 = \mathcal{S} - G_1$. T_1 and T_2 are obviously perfect phylogenies for G_1 and G_2. We can construct a subphylogeny for G_1 from T_1 as follows. For each $c \in \mathcal{D}(G_1)$ such that u_c is not the state of some species in G_1, let B_c be the set of all $b \in V(T_1)$ such that $b_c = u_c$. Let d be any node in $V(T_1) - B_c$ that is adjacent to a node in B_c. Such a d must exist, for otherwise we would have $a_c = u_c$ for every $\boldsymbol{a} \in G_1$, which contradicts the assumption that u_c is not the state of some species in G_1. We must have $d_c = a_c$ for some $\boldsymbol{a} \in G_1$, otherwise (C3) would be violated in T because we would have $a_c = b_c$ for some $\boldsymbol{b} \in T_2$, and the node $\boldsymbol{u}$ which is on the path between $\boldsymbol{a}$ and $\boldsymbol{b}$ in T is such that $u_c \neq a_c$. Now, set $b_c = d_c$ for all $\boldsymbol{b} \in B_c$. Since T satisfies (C3), for every $c \in \mathcal{M}(G_1)$, u_c equals the unique common state between G_1 and G_2. The resulting modification of T_1 is therefore a subphylogeny for G_1 with connection $\boldsymbol{u}$. An analogous construction can be used to obtain a subphylogeny for G_2. □

DEFINITION 3.3. *A cluster G is said to be* compatible *with a vector* $\boldsymbol{s}$ *if for every* $c \in \mathcal{M}(G)$, s_c *equals the unique common state for character c between G and* $\mathcal{S} - G$.

The following result demonstrates that a subphylogeny for a cluster can always be assembled from subphylogenies for c-clusters.

LEMMA 3.4. *Let G be a cluster. Then G has a subphylogeny if and only if there exist pairwise disjoint c-clusters* $G_1, \ldots, G_k$ *and a vector* $\boldsymbol{x}$ *such that* (i) *for every* $c \in \mathcal{M}(G)$, x_c *equals the* (*unique*) *common state for character c between G and* $\mathcal{S} - G$, (ii) $\cup_{i=1}^{k} G_i = G$, *and* (iii) *each* G_i *is compatible with* $\boldsymbol{x}$ *and has a subphylogeny.*

Proof. For the "if" part, let $T_1, \ldots, T_k$ be the subphylogenies for $G_1, \ldots, G_k$ with roots $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^k$. Clearly, the tree T consisting of a node for $\boldsymbol{x}$ and the trees $T_1, \ldots, T_k$ connected to $\boldsymbol{x}$ by edges $(\boldsymbol{x}^1, \boldsymbol{x}), \ldots, (\boldsymbol{x}^k, \boldsymbol{x})$ is a subphylogeny for G.

For the "only if " part, let T be a subphylogeny for G with connection $\boldsymbol{x}$. Without loss of generality, assume that all nodes in T are distinct. Let $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^k$ be the neighbors of $\boldsymbol{x}$ in T, and for $1 \leq i \leq k$ let T_k be the subtree of $T - \boldsymbol{x}$ containing $\boldsymbol{x}^i$ and let $G_i = V(T_i) \cap \mathcal{S}$. For each $c \in \mathcal{M}(G_i)$, x_c equals the unique common state between G_i and $\mathcal{S} - G_i$. This is because either this state is shared with some species in G_j, for some $j \neq i$, or it is shared with some species in $\mathcal{S} - G$. In either case, due to condition (C3), the value of x_c^i must equal the common state and, hence, G_i is compatible with $\boldsymbol{x}$. Also, as in the proof of Lemma 3.2, we can ensure that for every character c, the state of any $\boldsymbol{v} \in V(T_i)$ on character c will be that of some species in G_i on c, by altering unforced states if needed. Thus, T_i can be transformed into a subphylogeny for G_i. All that is left is to verify that each G_i is indeed a c-split; i.e., that $\mathcal{D}(G_i) \neq \emptyset$. Suppose $\mathcal{D}(G_i) = \emptyset$. Then we must have had $\boldsymbol{x}^i = \boldsymbol{x}$ in T, contradicting our earlier assumption that all nodes are distinct, since for every character c, there is one common character state between G and $\mathcal{S} - G$ and condition (C3) must be satisfied in T. □

To find a perfect phylogeny for $\mathcal{S}$, we shall rely on certain properties of subphylogenies which allow them to be combined into larger subphylogenies. These properties are discussed next.

LEMMA 3.5. *Let* G, G_1, G_2 *be clusters such that* $G = G_1 \cup G_2$ *and* $G_1 \cap G_2 = \emptyset$. *If* G_1 *and* G_2 *have subphylogenies, then there exists a subphylogeny T for G.*

Proof. Let T_1 and T_2 be subphylogenies for G_1 and G_2, respectively. Let $\boldsymbol{x}^1$ and $\boldsymbol{x}^2$ be the connections of T_1 and T_2. Let $\boldsymbol{x}$ be a node where for each $c \in \mathcal{M}(G)$, x_c equals the unique common state between G and $\mathcal{S} - G$ and for each $c \in \mathcal{D}(G)$, x_c equals x_c^1. Construct T by adding a node $\boldsymbol{x}$ and connecting the trees T_1 and T_2 to $\boldsymbol{x}$ by edges $(\boldsymbol{x}, \boldsymbol{x}^1)$ and $(\boldsymbol{x}, \boldsymbol{x}^2)$, respectively; $\boldsymbol{x}$ will be the connection of T. Since $\boldsymbol{x}$ has the necessary states to be a connection for a subphylogeny of G, it suffices to prove that T is a perfect phylogeny for G. For this, we need to verify that T satisfies conditions (C1)–(C3). Since T_1 and T_2 are subphylogenies, and for every character c, x_c is the state of some species in G_1 or G_2, it is clear that T satisfies (C1) and (C2) for G. To verify that T satisfies (C3), we must show that if $x_c^1 = x_c^2 = j$ for any character $c \in \mathcal{C}$, then $x_c = j$. This is trivially true when $c \in \mathcal{D}(G)$, since $x_c = x_c^1$. If $c \in \mathcal{M}(G)$, x_c equals the unique common state for character c between G and $\mathcal{S} - G$. We must have $x_c = x_c^1$ or $x_c = x_c^2$ because the species in G sharing the common character state with a species in $\mathcal{S} - G$ belongs to either G_1 or G_2. Hence, T satisfies (C3).

Note that if $\mathcal{D}(G_1) = \emptyset$, then, rather than adding an edge $(\boldsymbol{x}, \boldsymbol{x}^1)$, we can simply identify nodes $\boldsymbol{x}^1$ and $\boldsymbol{x}$. A similar situation arises when $\mathcal{D}(G_2) = \emptyset$. □

LEMMA 3.6. *Let G, G_1, G_2 be clusters such that $G = G_1 \cup G_2$ and $G_1 \cap G_2 = \emptyset$. Suppose that G_1 has a subphylogeny T_1 and that there exists a subphylogeny T for G with T_1 as a subtree at the connection $\boldsymbol{x}$ of T. Then if G_2 is not a c-cluster, the value of x_c on every $c \in \mathcal{D}(G)$ is forced.*

Proof. If G_2 is not a c-cluster, $\mathcal{M}(G_2) = \mathcal{C}$; i.e., for every $c \in \mathcal{C}$, there exists a common state between a species in G_2 and one in $\mathcal{S} - G_2$. Consider any character $c \in \mathcal{D}(G)$. We claim that G_1 and G_2 share a state on c. This implies that x_c is forced. To prove the claim, assume the contrary. Since $c \in \mathcal{M}(G_2)$, G_2 must share a state with $\mathcal{S} - G$ on character c. But this would imply that $c \in \mathcal{M}(G)$, a contradiction. □

4. Building a subphylogeny. The heart of our perfect phylogeny algorithm is a procedure SUBPHYLOGENY that determines whether a cluster G has a subphylogeny and, if so, constructs one. It assumes that for every c-cluster $G' \subset G$, a subphylogeny has been constructed, if one exists.

```
        Algorithm SUBPHYLOGENY($G$)
        begin
            if $|G| = 1$ then
(S1)            return the tree $T_G$ consisting of the single species $a \in G$
(S2)        for each c-cluster $G_1 \subset G$ such that $G_1$ has a subphylogeny $T_1$ do
                $G_2 = G - G_1$
                if $G_2$ is a c-cluster having a subphylogeny then
                    Use Lemma 3.5 to construct a subphylogeny $T_G$ for $G$
(S3)                return $T_G$
                else        /* $G_2$ is not a c-cluster */
                    Use Lemma 3.6 to compute the states of the connection $x$ of $T_G$
                    Initialize $T_G$ to consist of $x$ together with $T_1$ as its subtree
(S4)                for each c-cluster $H \subseteq G_2$ do
                        if $H$ has a subphylogeny $T_H$ and $H$ is compatible with $x$ then
                            $G_2 = G_2 - H$
                            Make $T_H$ a subtree of $x$ in $T_G$
(S5)                        if $G_2 = \emptyset$ then return $T_G$
                endif
            endfor
            return FAILURE
        end
```

LEMMA 4.1. *Let G be a cluster and suppose that for every c-cluster G' such that $G' \subset G$, we have determined whether G' has a subphylogeny and, if so, one has been constructed. Then, if G has a subphylogeny,* SUBPHYLOGENY(G) *constructs it. Otherwise, the procedure returns* FAILURE.

Proof. When $|G| = 1$, a node for the single species $s \in G$ is indeed a subphylogeny for G. Hence, the tree returned in (S1) is correct.

Suppose $|G| > 1$ and that G has a subphylogeny T_G with connection $\boldsymbol{x}$. Then there must exist a c-cluster $G_1 \subseteq G$ having a subphylogeny T_1 such that T_1 is a subtree in some subphylogeny T_G of G. At some point during the execution of **for** loop (S2), we will consider one such G_1. If $G_2 = G - G_1$ is a c-cluster having a subphylogeny, then, by Lemma 3.5, (S3) returns a subphylogeny for G.

If G_2 is not a c-cluster, then, by Lemma 3.6, the states of the connection $\boldsymbol{x}$ are completely determined and, by Lemma 3.4, there exists a collection of pairwise disjoint c-clusters having subphylogenies such that their union is G and each c-cluster in the collection is compatible with $\boldsymbol{x}$. In the **for** loop (S4), a subphylogeny for a c-cluster $H \subseteq G$ is added to the current T_G only if H is compatible with $\boldsymbol{x}$ and every species in H is not yet in T_G, and a tree is returned only when the union of all the c-clusters added is $G - G_1$; i.e., $G_2 = \emptyset$. Therefore, any tree T_G returned in (S5) is a subphylogeny for G. We now argue that if there exists a subphylogeny for G with T_1 as a subtree, then one such subphylogeny will be constructed by the loop (S4) and G_2 will become empty. We shall do this by showing that if there is such a subphylogeny for G, then at the beginning of each iteration of loop (S4) there is always at least one c-cluster $H' \subseteq G_2$ compatible with $\boldsymbol{x}$ that has not yet been considered.

By Lemma 2.2, if G has a subphylogeny, then for every $J \subseteq G$ the set of species $J \cup \{\boldsymbol{x}\}$ has a perfect phylogeny. We claim that there exists a c-cluster $J' \subseteq J$ such that J' has a subphylogeny. To prove this, suppose T_J is a perfect phylogeny for $J \cup \{\boldsymbol{x}\}$, let $\boldsymbol{y}$ be any neighbor of $\boldsymbol{x}$ in T_J, and let J' be the set of species in the subtree T_y of $T_J - \boldsymbol{x}$ containing $\boldsymbol{y}$. Clearly, J' is a cluster; moreover, J' is a c-cluster since we can assume that all the nodes in T_J are distinct. Furthermore, T_y can be modified to obtain a subphylogeny for J' as in the proof of Lemma 3.2.

In particular, if we take $J = G_2$ in the above argument, we have that in each iteration of (S4), there exists a c-cluster $J' \subseteq G_2$ such that J' has a subphylogeny. Since at the beginning of each iteration, for all c-clusters J'' considered up to this point $J'' \not\subseteq G_2$, either H itself is a subset of G_2 and has a subphylogeny, or some yet-to-be-considered c-cluster H' is a subset of G_2 and has a subphylogeny.

SUBPHYLOGENY returns FAILURE only if no choice of G_1 led to the construction of a subphylogeny for G. In this event, there was certainly no subphylogeny for G. □

Running time of SUBPHYLOGENY. SUBPHYLOGENY(G) considers each of the $O(2^r m)$ c-clusters G_1 such that $|G_1| < |G|$. For each such c-cluster, it verifies that $G_1 \subset G$, which can be done in $O(n)$ time. With a particular G_1, the algorithm goes through $O(2^r m)$ c-clusters, checking in $O(n + m)$ time whether they are subsets of G_2 that are compatible with $\boldsymbol{x}$. The total time taken by SUBPHYLOGENY is therefore $O(2^{2r}(nm^2 + m^3))$.

5. Building a perfect phylogeny.

We now describe the algorithm PHYLOGENY, which constructs a perfect phylogeny for $\mathcal{S}$, if it has one. The algorithm first tries to determine if one of the $O(2^{r-1} \cdot m)$ c-splits is of type I. If there is a type I c-split (G_1, G_2) where $\boldsymbol{s}$ is a connecting species, the algorithm recursively attempts to construct perfect phylogenies T_1 and T_2 for $G_1 \cup \{\boldsymbol{s}\}$ and $G_2 \cup \{\boldsymbol{s}\}$. As stated earlier, if one or both of the latter sets has no perfect phylogeny then, by Lemma 2.2, neither does $\mathcal{S}$. Otherwise, a perfect phylogeny for $\mathcal{S}$ is obtained by identifying the nodes for $\boldsymbol{s}$ in T_1 and T_2.

If there is no type I c-split then, by Lemma 2.6, none of the species appears as an internal node in any perfect phylogeny for $\mathcal{S}$. In this case, PHYLOGENY considers each possible c-cluster G such that $|G| \leq n - 1$ and attempts to build a subphylogeny for it using SUBPHYLOGENY. It then uses this information to build a perfect phylogeny for $\mathcal{S}$, if possible. The steps of PHYLOGENY are as follows.

```
 Algorithm PHYLOGENY($\mathcal{S}$)
begin
    if $|\mathcal{S}| = 1$ then
        return the tree $T$ consisting of the single species $a \in \mathcal{S}$
    if there exists a type I c-split $(G_1, G_2)$ then
        Let $s$ be the connecting species
```

```
            Call PHYLOGENY($G_1 \cup \{s\}$) and PHYLOGENY($G_2 \cup \{s\}$)
            if both calls succeed then
                Combine the resulting trees into a perfect phylogeny for $\mathcal{S}$
            else return FAILURE
        else                    /* All c-splits are of type II */
            $size = 1$
            while $size \leq n - 1$ do
(P1)            for each c-cluster $G$ such that $|G| = size$ do
                    Call SUBPHYLOGENY($G$)
                    if $G$ has a subphylogeny $T_G$, record $T_G$ and its connection
                endfor
                $size = size + 1$
            endwhile
            Pick any $s \in \mathcal{S}$
            if $G = \mathcal{S} - \{s\}$ has a subphylogeny $T_G$ then
                let $x$ be the connection of $T_G$
(P2)            return the tree obtained by adding a node $s$ and the edge $(s, x)$ to $T_G$
(P3)        else return FAILURE
        endif
    end
```

THEOREM 5.1. PHYLOGENY *correctly determines whether or not $\mathcal{S}$ has a perfect phylogeny and, if so, constructs one.*

Proof. The correctness of the algorithm hinges on how it deals with the case where no c-split is of type I. We first note that by the proof of Lemma 3.2, and because for any $s \in \mathcal{S}$ the tree consisting of node s is a subphylogeny for $\{s\}$, if a tree is returned by our algorithm in (P2), then the tree is a perfect phylogeny for $\mathcal{S}$. Thus, it suffices to argue that PHYLOGENY will never return FAILURE if $\mathcal{S}$ has a perfect phylogeny.

If $\mathcal{S}$ has a perfect phylogeny and all c-splits are of type II, then every $s \in \mathcal{S}$ will be a leaf in any perfect phylogeny of $\mathcal{S}$. Hence, both $\{s\}$ and $\mathcal{S} - \{s\}$ must be c-clusters. As we noted above, a subphylogeny for $\{s\}$ is s itself, while the subphylogeny of $\mathcal{S} - \{s\}$ must have been constructed in some iteration of (P1). Therefore, (P2) will not return FAILURE. □

Running time of PHYLOGENY. PHYLOGENY spends $O(2^r nm^2)$ time generating c-clusters and testing each of these to find out whether it is of type I. It is clear that, in the worst case, the running time of PHYLOGENY is dominated by the time required to deal with the case where all c-clusters are of type II. SUBPHYLOGENY is applied to each of the $O(2^r m)$ c-clusters, which requires $O(2^{3r}(nm^3 + m^4))$ total time. Hence, the running time of PHYLOGENY is $O(2^{3r}(nm^3 + m^4))$ as well.

6. Concluding remarks. Our algorithm uses dynamic programming to construct perfect phylogenies by working from the bottom up. One can use *memoization* (a technique described in some detail in pp. 312–314 of [4]) to obtain an equivalent top-down recursive algorithm with the same running time. Such a procedure has been proposed to us by E. L. Lawler (personal communication).

Algorithm PHYLOGENY can be modified to work correctly and within the same time bounds even if instances with type I c-splits are not treated separately. However, in practice, exploiting the presence of such splits may tend to reduce the running time of the algorithm.

Kannan and Warnow (personal communication) have discovered a clever way to reduce the running time of our algorithm by a factor of 2^r. Their technique speeds up step (S4) of SUBPHYLOGENY by providing a way to determine in $O(nm)$ time whether there exists a subphylogeny for G having a subphylogeny for a given c-cluster G_1 as a subtree. Even with this improvement, the algorithms presented in [16] and [7] are faster than ours for the cases where $r \leq 4$ and $r \leq 3$, respectively. It is an open problem whether our algorithm can be improved to match those bounds on those special cases.

Other methodologies for reconstructing phylogenies have been proposed in the past; these have been ably surveyed by Felsenstein [10]. We shall limit ourselves here to discussing one problem that is closely related to perfect phylogeny: the *Steiner tree problem in phylogeny* [12], [13]. A Steiner tree for a set of species $\mathcal{S}$ is a tree T satisfying (C1) and (C2). Obviously, a perfect phylogeny for $\mathcal{S}$ is also a Steiner tree for $\mathcal{S}$. The length of T is the sum of the lengths of its edges, where the length of an edge $(\boldsymbol{u}, \boldsymbol{v}) \in E(T)$ is the Hamming distance between $\boldsymbol{u}$ and $\boldsymbol{v}$ (i.e., the number of characters in which $\boldsymbol{u}$ and $\boldsymbol{v}$ differ). The Steiner tree problem in phylogeny is to compute a minimum length Steiner tree for a given set of species. Minimum length Steiner trees satisfy the *parsimony* criterion [10], because they give phylogenies in which species evolve with the least number of character changes. The following theorem establishes a relationship between the perfect phylogeny problem and the Steiner tree problem in phylogeny.

THEOREM 6.1. *$\mathcal{S}$ has a perfect phylogeny if and only if the minimum length of a Steiner tree for $\mathcal{S}$ is $\sum_{i=1}^{m}(r_i - 1)$.*

Proof. Suppose T is a Steiner tree. It follows from the definition that the length of T is the sum of contributions of the individual characters, where the contribution of character c to the length of T is the number of edges $(\boldsymbol{u}, \boldsymbol{v}) \in E(T)$ such that $u_c \neq v_c$. Since there are at least r_c species which differ from each other on any $c \in \mathcal{C}$, the contribution of character c to the length of T must be at least $r_c - 1$, implying that the length of any Steiner tree for $\mathcal{S}$ is at least $\sum_{i=1}^{m}(r_i - 1)$.

A Steiner tree T for $\mathcal{S}$ has length *exactly* $\sum_{i=1}^{m}(r_i - 1)$ if and only if for every $c \in \mathcal{C}$, T can be partitioned into exactly r_c subtrees $T_1, \ldots, T_{r_c}$ such that $u_c = i$ for every $\boldsymbol{u} \in T_i$. If such a partition is possible, T will satisfy (C3), in addition to (C1) and (C2), and must be a perfect phylogeny for $\mathcal{S}$. □

It does not seem possible to generalize our algorithm in any straightforward way to produce minimum length Steiner trees for sets of species. To illustrate the difficulty in doing so, note that whereas for binary characters, the perfect phylogeny problem can be solved in time linear in the size of the input, the Steiner tree problem in phylogeny remains NP complete [13].

Acknowledgments. We thank Martin Farach, Sampath Kannan, and Tandy Warnow for their encouragement; Mike Steel for bringing his results to our attention; and the referees for their useful suggestions. Special thanks are due to Eugene Lawler, who carefully read our manuscript and suggested numerous improvements in terminology and exposition.

REFERENCES

[1] R. AGARWALA, D. FERNÁNDEZ-BACA, AND G. SLUTZKI, *Fast algorithms for inferring evolutionary trees*, In Proceedings of the 30th Allerton Conference on Comm., Control, and Comput., 1992, pp. 594–603.

[2] H. BODLAENDER, M. FELLOWS, AND T. WARNOW, *Two strikes against perfect phylogeny*, In Proceedings of the 19th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, Springer Verlag, Berlin, New York, 1992, pp. 273–283.

[3] H. BODLAENDER AND T. KLOKS, *A simple linear time algorithm for triangulating three-colored graphs*, In Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, 1992, pp. 415–423.

[4] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

[5] J. CAMIN AND R. SOKAL, *A method for deducing branching sequences in phylogeny*, Evolution, 19 (1965), pp. 311–326.

[6] W. H. E. Day and D. Sankoff, *Computational complexity of inferring phylogenies by compatibility*, Syst. Zool., 35 (1986), pp. 224–229.

[7] A. Dress and M. Steel, *Convex tree realizations of partitions*, Appl. Math. Lett., 5 (1992), pp. 3–6.

[8] G. F. Estabrook, C. S. Johnson Jr., and F. R. McMorris, *An idealized concept of the true cladistic character*, Mathematical Biosciences, 23 (1975), pp. 263–272.

[9] G. F. Estabrook, *Cladistic methodology: A discussion of the theoretical basis for the induction of evolutionary history*, Annual Review of Ecology and Systematics, 3 (1972), pp. 427–456.

[10] J. S. Felsenstein, *Phylogenies from molecular sequences: Inference and reliability*, Annual Reviews of Genetics, 22 (1988), pp. 521–565.

[11] W. M. Fitch, *Aspects of molecular evolution*, Annual Reviews of Genetics, 7 (1973), pp. 343–380.

[12] L. R. Foulds, *Maximum savings in the Steiner problem in phylogeny*, J. Theoret. Biol., 107 (1983), pp. 471–474.

[13] L. R. Foulds and R. L. Graham, *The Steiner problem in phylogeny is NP-complete*, Adv. in Appl. Math., 3 (1982), pp. 43–49.

[14] D. Gusfield, *Efficient algorithms for inferring evolutionary trees*, Networks, 21 (1991), pp. 19–28.

[15] S. Kannan and T. Warnow, *Triangulating three-colored graphs*, SIAM J. Discrete Math., 5 (1992), pp. 249–258.

[16] S. Kannan and T. Warnow, *Inferring evolutionary history from DNA sequences*, in Proceedings of the 31st Annual Symposium on the Foundations of Computer Science, 1990, St. Louis, MO, pp. 362–378.

[17] W. J. Le Quesne, *A method of selection of characters in numerical taxonomy*, Syst. Zool., 18 (1969), pp. 201–205.

[18] W. J. Le Quesne, *Further studies based on the uniquely derived character concept*, Syst. Zool., 21 (1972), pp. 281–288.

[19] F. R. McMorris, T. Warnow, and T. Wimer, *Triangulating vertex colored graphs*, In Proceedings of the 4th Annual Symposium on Discrete Algorithms, 1993, Austin, TX, pp. 120–127.

[20] M. A. Steel, *The complexity of reconstructing trees from qualitative characters and subtrees*, J. Classification, 9 (1992), pp. 91–116.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1225–1230, December 1994

007

PRIORITY QUEUES AND PERMUTATIONS*

M. D. ATKINSON† AND ROBERT BEALS‡

Abstract. A priority queue transforms an input permutation σ of some set of size n into an output permutation τ. The set R_n of such related pairs (σ, τ) is studied. Efficient algorithms for determining $s(\tau) = |\sigma : (\sigma, \tau) \in R_n|$ and $t(\sigma) = |\tau : (\sigma, \tau) \in R_n|$ are given, a new proof that $|R_n| = (n+1)^{n-1}$ is given, and the transitive closure of R_n is found.

Key words. priority queue, permutation, enumeration

AMS subject classifications. primary 68R05; secondary 68P05, 05A15

1. Introduction. The topic studied in this paper was suggested by the beautiful and combinatorially rich theory of permutations that can be generated by a stack. In this theory an input sequence of n distinct elements, $x_1, x_2, \ldots, x_n$, is presented to a stack and is subjected to a series of *push* and *pop* operations; each push operation pushes the next input element on the stack and each pop operation removes the top element of the stack and places it in an output stream. When the entire input has been consumed and the stack is empty, the input sequence has been converted into an output sequence $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$ that is a permutation of the input sequence. The permutation π that arises in this way depends only on the series of push and pop operations and, in particular, is independent of the input sequence. It is well known that there are c_n such *stack permutations,* one for each push–pop sequence, where

$$c_n = \frac{\binom{2n}{n}}{n+1}$$

is the nth Catalan number. There are several interesting correspondences between stack permutations and other combinatorial objects (for example, binary trees, triangulations of a polygon, well-formed bracket sequences; see [3] and the references cited therein).

We shall investigate the analogous theory where a priority queue replaces a stack. In other words the pop operation "delete the most recently inserted" is replaced by "delete the smallest." We use the terms *Insert* and *Delete-Minimum* rather than push and pop and we shall call a series of n Inserts and n Delete-Minimums a *priority queue computation.* There are c_n different priority queue computations, each of which converts an input sequence (of n distinct elements) into an output sequence. But, unlike the case of stacks, it is no longer true that the permutational effect is independent of the input sequence nor that all inputs can be permuted in the same number of ways.

To handle this greater complexity we define a relation R_n called *allowability* on the set of sequences of length n by the rule $(\sigma, \tau) \in R_n$ if there exists a priority queue computation which transforms the input sequence σ into the output sequence τ. The elements of R_n will be called *allowable pairs.* It was shown in [1] that there are $(n+1)^{n-1}$ allowable pairs of permutations on a fixed set of size n and this result is a strong hint that an interesting combinatorial theory awaits investigation.

It is simple to recognize whether a pair of sequences (σ, τ) is allowable. One just constructs a suitable priority queue computation. If such a computation exists, it may not be unique, but it is easy to see that there is always a "natural" computation in which elements

*Received by the editors November 30, 1992; accepted for publication July 20, 1993.

†Department of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St. Andrews, Fife KY16 9SS, Scotland.

‡Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403.

are output as soon as possible (that is, if x is the next element to output, and is present in the priority queue, then it should be output before further input elements are inserted). This observation results in the following algorithm:

```
$Q$ := empty priority queue
$i := 1$; $j := 1$
while $j \leq n$ do
        while $\tau_j \notin Q$ do
                insert($\sigma_i$)
                $i := i + 1$
        endwhile
        if min($Q$) $\neq \tau_j$ then return (false) else
                deleteMin; $j := j + 1$
        endif
endwhile
return(true).
```

The time complexity of this algorithm depends, of course, on the implementation of the priority queue operations. With a heap-based implementation it would be $O(n \log n)$, whereas if we regarded Insert and Delete-Minimum as atomic constant time operations it would be $O(n)$.

In the next section of the paper we take up rather more interesting algorithmic questions by giving efficient methods for the two complementary problems:

(1) Given an output τ, find the number of allowable pairs (σ, τ).

(2) Given an input σ, find the number of allowable pairs (σ, τ).

Then, in §3 we concentrate more on combinatorial questions. We give a one-to-one correspondence between allowable pairs and labeled trees from which we obtain another proof of the main result of [1]. Next we give a result about the average number of outputs for a random priority queue computation. Finally we characterize the transitive closure of the allowability relation and discuss the connection with serial networks of priority queues.

2. The number of inputs and outputs. In this section we consider the computation of the following two quantities: the number $s(\tau) = |\sigma : (\sigma, \tau) \in R_n|$ of allowable pairs having a fixed output τ, and the number $t(\sigma) = |\tau : (\sigma, \tau) \in R_n|$ of allowable pairs having a fixed input σ.

On the surface $t(\sigma)$ seems to be the more natural quantity to calculate since it enumerates the different ways that a priority queue can process a particular input, while $s(\tau)$ enumerates the different starting points that can give rise to some fixed result. Despite this, it turns out that the numbers $s(\tau)$ are rather easier to compute and have more obvious properties (for example, they are always divisors of $n!$). Indeed, in [1], an algorithm of average time complexity $O(n \log n)$ was given for computing $s(\tau)$. Here we shall give an algorithm for this problem whose worst case complexity is $O(n)$. We turn then to the problem of computing $t(\sigma)$. Although we have been unable to find a comparably efficient algorithm, we are at least able to place the problem in the complexity class $\mathcal{P}$ by giving a dynamic programming algorithm of time complexity $O(n^4)$.

For a given output sequence $\tau = [\tau_1, \ldots, \tau_n]$ we let τ_0 be any element greater than those occurring in the remainder of τ and define

$$b(i) = \max\{j : 1 \leq j < i, \tau_j > \tau_i\}.$$

The following lemma was proved in [1].

LEMMA 2.1.

$$s(\tau) = \prod_{i=1}^{n} (i - b(i)).$$

Thus, once $b(1), b(2), \ldots, b(n)$ have been calculated, $s(\tau)$ can be found in $O(n)$ steps. The obvious algorithm for computing $b(1), b(2), \ldots, b(n)$ is as follows.

```
$i := 0$
*repeat*
    $i := i + 1$; $j := i - 1$
    *while* $\tau_j < \tau_i$ *do* $j := j - 1$
    $b(i) := j$
*until* $i = n$.
```

It requires quadratic time in the worst case, since to compute $b(i)$ as many as i values of τ may need to be examined. A simple observation improves the computation time. Suppose that when computing $b(i)$, we tested the element τ_j (with $j < i$) and found that $\tau_j < \tau_i$. Then none of $\tau_{j-1}, \tau_{j-2}, \ldots, \tau_{b(j)+1}$ need be compared with τ_i and the next test can be whether $\tau_{b(j)} < \tau_i$. In other words the statement $j := j - 1$ may be replaced by $j := b(j)$.

The new algorithm runs in linear time: the statement $j := b(j)$ cannot be executed more than once with the same value of j. Consequently $s(\tau)$ can be found from τ in $O(n)$ steps.

We now consider the second computation: finding $t(\sigma)$. The following result was proved in [1].

LEMMA 2.2. *Let σ be some input sequence expressed in the form $\alpha m \beta$ where m is the maximal symbol. Suppose $\beta = b_1 b_2 \cdots b_r$ and $\alpha_i = \alpha b_1 \cdots b_i$, $\beta_i = b_{i+1} \cdots b_r$, then*

$$t(\sigma) = \sum_{i=0}^{r} t(\alpha_i) t(\beta_i).$$

We make this lemma the basis of a dynamic programming algorithm. For convenience assume that σ is a permutation of $\{1, 2, \ldots, n\}$. Let $\sigma^{(m)}$ denote the permutation obtained from σ by deleting all the symbols $m + 1, \ldots, n$ and let $\Sigma^{(m)}$ denote the set of substrings of $\sigma^{(m)}$.

We can compute $t(\theta)$ for all $\theta \in \Sigma^{(m)}$ for the values $m = 0, 1, 2, \ldots, n$ in turn. The empty string λ is the only member of $\Sigma^{(0)}$ and $t(\lambda) = 1$. Suppose that $m \geq 1$ and $t(\theta)$ is known for all $\theta \in \Sigma^{(m-1)}$. A string ϕ of $\Sigma^{(m)}$ is either in $\Sigma^{(m-1)}$ (in which case $t(\phi)$ will be known) or it has the form $\alpha m \beta$. In the latter case we may compute $t(\alpha m \beta)$ in linear time using the formula of the previous lemma since all the strings required in the calculation belong to $\Sigma^{(m-1)}$. The total time required for the whole computation is

$$O\left(n \times \sum_{m=1}^{n} |\Sigma^{(m)}|\right) = O(n^4).$$

3. Combinatorial results. The number of allowable pairs was proved in [1] to be $(n + 1)^{n-1}$ using partially ordered sets and one of Abel's summation formulae. Here we give another proof which depends on establishing a 1-1 correspondence between allowable pairs and labeled trees.

Let (σ, τ) be an allowable pair on n symbols and let m be the maximal symbol. Suppose $\tau = \alpha m \beta$. At the point that m is output, the priority queue is empty. Therefore m and all the symbols of α occur in σ earlier than all the symbols of β. Let γ,δ be the symbols of α,

β, respectively, in order of their occurrence within σ. Then σ has the form $\gamma_{(i,m)}\delta$ where $\gamma_{(i,m)}$ denotes the result of inserting m within γ after the symbol i (i is given the conventional name "root" if m is inserted at the beginning of γ). Clearly (γ, α) and (δ, β) are allowable pairs.

Thus (σ, τ) has given rise to allowable pairs (γ, α), (δ, β) and a symbol i (one of the symbols of α or "root"). Conversely, (γ, α), (δ, β), and i define a unique allowable pair (σ, τ) by reversal of this construction.

We can now associate, with any allowable pair (σ, τ), a tree on $n+1$ symbols ("root" and the n symbols being permuted). The construction is inductive. If $n = 0$ the tree is a single node called "root." If $n > 0$ we find (γ, α), (δ, β), and i as above. The trees for (γ, α), (δ, β) exist by induction and have roots "root1" and "root2." The tree for (σ, τ) is obtained by letting "root1" be the new "root," "root2" be labeled as m, and joining nodes i and m.

The construction is obviously reversible since the parent of the maximal node in a tree defines i and removal of the branch between i and m defines the trees for the allowable pairs (γ, α), (δ, β).

THEOREM 3.1 (Atkinson–Thiyagarajah [1]). *The number of allowable pairs on a set of size n is $(n+1)^{n-1}$.*

Proof. By Cayley's theorem the number of labeled unrooted trees on $n+1$ nodes is $(n+1)^{n-1}$. The trees defined above are rooted (which would increase their number by a factor $n+1$) but the root is labeled with a fixed symbol "root" (which decreases their number by a factor $n+1$). The result now follows from the correspondence above. □

We now consider the number of outputs that a fixed priority queue computation can produce as the input varies over all permutations of an n-element set. Clearly, there is considerable variation. If we let $\mathcal{I}$ and $\mathcal{D}$ denote the operations Insert and Delete-Minimum, respectively, then priority queue computations can be represented by "well-formed" words in these two symbols. For example, the computation $(\mathcal{ID})^n$ simply copies the input to the output so all $n!$ outputs are achievable. On the other hand, $\mathcal{I}^n\mathcal{D}^n$ only produces that output whose elements are in ascending order. For notational simplicity we shall assume that all inputs and outputs are permutations of $\{1, 2, \dots, n\}$. Let $k(w)$ denote the number of outputs that can be generated by the well-formed word w.

LEMMA 3.2.

(1) *If u, v are well formed of length $2a$, $2b$, then*

$$k(uv) = k(u)k(v)\binom{a+b}{a}.$$

(2) *If u is well formed, then $k(\mathcal{I}u\mathcal{D}) = k(u)$.*

Proof. First, consider $k(uv)$. The priority queue in the computation uv is empty just before the $(a+1)$th symbol of the input is read. So the first a symbols of the output are a permutation of the first a symbols of the input. The first a symbols may be chosen in $\binom{a+b}{a}$ ways and, once chosen, may be permuted by u in $k(u)$ ways, after which the remaining b symbols may be permuted in $k(v)$ ways. The first part now follows.

Now consider $k(\mathcal{I}u\mathcal{D})$. Note first that if τ is an output of the priority queue computation u (arising, say, from the input σ) then τn is an output from the computation $\mathcal{I}u\mathcal{D}$ (clearly, it is the result of processing the input $n\sigma$). On the other hand, any output from $\mathcal{I}u\mathcal{D}$ must have the form τn for some τ. This is because, for the computation $\mathcal{I}u\mathcal{D}$, the priority queue cannot become empty at any intermediate step and so n cannot be deleted from it as smallest element until the end of the computation. To complete the proof, it is only necessary to prove that τ is an output of the priority queue computation u. Let ρ be an input to the computation $\mathcal{I}u\mathcal{D}$ which gives the output τn. If n is not the first symbol of ρ then we can write $\rho = \alpha m \beta n \gamma$, where m is the maximum symbol that precedes n in ρ. When this input is processed by $\mathcal{I}u\mathcal{D}$, the symbol m will remain in the priority queue after it is inserted at least until n is inserted (otherwise the priority queue would become empty before the computation terminates). It therefore follows that the input $\alpha n\beta m\gamma$ will result in the same output as ρ. By repetition of this principle, we obtain an input of the form $n\sigma$ which generates the output τn. It is apparent that the priority queue computation u transforms σ into τ as required. □

COROLLARY 3.3. *For every priority queue computation u, $k(u)$ divides $n!$.*

We have already remarked that $k(u)$ varies considerably as u varies over the possible priority queue computations. Despite this, it is possible to compute its average value defined as

$$k_n = \sum_u k(u)/c_n,$$

where the summation is over all priority queue computations with n Inserts and n Delete-Minimums.

THEOREM 3.4. $k_n = (n+1)!/2^n$.

Proof. Let $l_n = \sum_u k(u)$ so that $k_n = l_n/c_n$. Every priority queue computation can be expressed in the form $u = \mathcal{I}v\mathcal{D}w$, where v, w are themselves priority queue computations and are uniquely determined by u. Since the length of v can be any even integer from 0 to $2n-2$ we have

$$l_n = \sum_u k(u) = \sum_{i=0}^{i=n-1} \sum_v \sum_w k(\mathcal{I}v\mathcal{D}w),$$

where the second and third summations are over all priority queue computations v, w which process inputs of length i, $n-i-1$, respectively. Thus, by the previous lemma,

$$l_n = \sum_{i=0}^{i=n-1} \sum_v \sum_w k(v)k(w)\binom{n}{i+1} = \sum_{i=0}^{i=n-1} l_i l_{n-i-1}\binom{n}{i+1}.$$

Put $h_n = l_n/n!$. Then $h_n = \sum_{i=0}^{i=n-1} h_i h_{n-i-1}/(i+1)$. This recurrence can be solved by introducing the generating function

$$H(x) = \sum_{r=0}^{r=\infty} h_r x^r$$

which is easily seen to satisfy the integral equation

$$H(x)\int H(x)dx = H(x) - 1,$$

from which it follows that $H(x) = 1/\sqrt{1-2x}$ and the result follows by expanding the generating function. □

Finally we consider the transitive closure of the allowability relation R_n. One motivation for this is Tarjan's paper [5] where the permutations obtainable from a series network of stacks are considered. In a related work, Pratt [4] considers permutations computable by double-ended queues, parallel stacks, and parallel queues. The corresponding scenario for us is a series network of priority queues $P_1, P_2, \ldots, P_k$ on which the following operations are allowed:

(1) Insert, which transfers the next element of the input sequence into P_1,

(2) Move(i) with $1 \le i < k$, which transfers the minimal element of P_i into P_{i+1},

(3) Delete-Minimum, which appends the minimal element of P_k onto the output sequence.

Let ${R_n}^k$ denote the k-fold composition of R_n with itself. Then it is easy to verify that ${R_n}^k$ is precisely the set of (input, output) pairs associated with a series network of k priority queues.

The *weak order* W_n on the set of all permutations on n symbols is defined by $(\sigma, \tau) \in W_n$ if and only if every pair of symbols of σ, which are in increasing order, are also in increasing order in τ. The weak order is an important tool for the study of geometric and combinatorial properties of the symmetric group; its properties are discussed at length in [2]. It is clear that $R_n \subseteq W_n$ and, since W_n is transitively closed, ${R_n}^k \subseteq W_n$ for all k.

THEOREM 3.5. ${R_n}^{n-2} \neq {R_n}^{n-1} = W_n$.

Proof. For $(\sigma, \tau) \in W_n$ let $\rho(\sigma, \tau)$ be the smallest integer k such that all but the leftmost k symbols of σ and τ agree (that is, the rightmost $n-k$ symbols agree). Observe that ρ is never equal to 1. In order to show that ${R_n}^{n-1} = W_n$ it suffices to show that for any $(\sigma, \tau) \in W_n$ with $\sigma \neq \tau$, there exists a σ' such that all of the following hold:

(1) $(\sigma, \sigma') \in R_n$,

(2) $(\sigma', \tau) \in W_n$,

(3) $\rho(\sigma', \tau) < \rho(\sigma, \tau)$.

To see this, write σ as $\alpha x \beta \gamma$ and τ as $\delta x \gamma$ such that $|\gamma| = n - \rho(\sigma, \delta)$. Since $(\sigma, \tau) \in W_n$, it follows that x must be larger than all symbols in β. Therefore, $\sigma' = \alpha\beta x \gamma$ satisfies condition (1) above (in fact a priority queue can input σ and output σ' by only placing two symbols, x and the current input symbol, in the queue at one time). Also, σ' satisfies condition (2) because any pair of symbols in σ' appear either in the same order as in σ, or the same order as in τ. Finally, since σ' and τ both end with $x\gamma$, condition (3) is satisfied. Since ρ cannot have the value 1, the sequence $\sigma', \sigma'', \dots$ must reach τ in at most $n-1$ steps, proving ${R_n}^{n-1} = W_n$.

To complete the proof, we prove that the pair

$$([n, n-1, \dots, 2, 1], [1, n, n-1, \dots, 3, 2]),$$

which plainly lies in W_n, does not belong to ${R_n}^{n-2}$. Suppose that it were possible to transform $[n, n-1, \dots, 2, 1]$ into $[1, n, n-1, \dots, 3, 2]$ by a series network of $n-2$ priority queues. When the final element of the input (symbol 1) has been placed in priority queue P_1 the other $n-1$ symbols must all be in the $n-2$ priority queues of the network since none can be output yet. One of the priority queues must therefore contain at least two of the symbols in $\{2, 3, \dots, n\}$. Clearly this is impossible, since these two symbols would then have to be output eventually in increasing order. □

Therefore, W_n describes the relation between input and output permutations for a series network of k priority queues for any $k \geq n-1$.

Acknowledgment. We thank Katherine Anderson and Murali Thiyagarajah for several useful conversations during the course of this work, and Gerald Ostheimer for the insight leading to the $O(n)$ algorithm for computing $s(\tau)$.

REFERENCES

[1] M. D. ATKINSON AND M. THIYAGARAJAH, *The permutational power of a priority queue*, BIT, 33 (1993), pp. 2–6.

[2] A. BJÖRNER, *Orderings on coxeter groups*, in Proceedings of Conference on Combinatorics and Algebra, American Mathematical Society, Providence, RI, 1983.

[3] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 1992.

[4] V. PRATT, *Computing permutations with double-ended queues, parallel stacks, and parallel queues*, in Proceedings of the 5th ACM STOC, ACM Press, New York, 1973, pp. 268–277.

[5] R. E. TARJAN, *Sorting using networks of queues and stacks*, J. ACM, 19 (1972), pp. 341–346.

SIAM J. COMPUT.
Vol.23, No.6, pp. 1231–1252, December 1994

008

A MODEL FOR ASYNCHRONOUS SHARED MEMORY PARALLEL COMPUTATION*

NAOMI NISHIMURA†

Abstract. Traditional theoretical shared memory parallel models have been based on a number of assumptions which simultaneously simplify solutions to problems and distance the models from actual parallel machines. One such assumption is that processors work together in a synchronous fashion. Recent work has focused on finding a model that captures the essence of computation by processors communicating asynchronously through shared memory. In this paper, a general framework and set of criteria used to analyze these models, including the complexity analysis of several fundamental algorithmic paradigms, are considered. A general asynchronous model is introduced and how it satisfies these criteria is demonstrated. In this model, $O(\log p)$ algorithms are demonstrated for solving p-input versions of the problems of AND, OR, parity, maximum, minimum, and list ranking. To handle list ranking, a technique of analyzing algorithms is developed in which the set of tasks that are to be executed depends on the processor schedules.

Key words. asynchrony, parallel computation, models of parallel computation, computational complexity, parallel algorithms

AMS subject classifications. 68Q05, 68Q10, 68Q22, 68Q25

1. Introduction.

As our understanding of the parallel random access machine (PRAM) [7], [10], [17] deepens, it becomes possible to extend the notion of a shared memory model to capture additional features of actual multiprocessor architectures. Implicit in the definition of a PRAM is the existence of a global clock, by means of which processors are able to synchronize actions and perform in lock-step. At each tick of the global clock, or *timestep* of a PRAM, each processor performs a local operation or accesses a shared memory cell. The synchronization of all processors with each other (the execution of a *synchronization barrier*) can be viewed as a PRAM primitive that requires time no greater than the time to execute a single processor step. In reality, however, synchronization of parallel processors itself can be a time-consuming task. Since synchronization is not necessarily built into the architecture, it is desirable to have algorithms that do not depend on the precise timing of steps taken by the various processors. The actual speeds of processors may vary according to operating system interference, contention for the network and memory, user requests, and disparities between the numbers of cycles required for different instructions. Similar arguments can be made with respect to asynchronous behavior of processes sharing processors; in the remainder of this paper we model processes by processors. Although the absence of asynchrony is only one of the impractical assumptions on which a PRAM is based, we initially remove only this assumption in order to isolate the contribution of synchronization and to keep problems tractable.

In order to implement a PRAM algorithm directly on an underlying asynchronous architecture, all processors must be synchronized after each PRAM step. Not only is this synchronization costly in terms of time, but it can also hide the true asynchronous complexity of the problem. To better understand the realities of asynchronous parallel computation, we begin, in §2, by outlining criteria for an asynchronous shared memory model. Of particular interest is the choice of the underlying synchronization primitive, viewed as a parameter of the model, and the choice of complexity measure. The measure is designed to reflect elapsed time, including effects of fluctuations in processor speeds and load balance; this is demonstrated with respect to three basic parallel algorithm paradigms.

*Received by the editors September 23, 1991; accepted for publication (in revised form) August 24, 1992. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1 (nishi@plg.uwaterloo.ca).

In §3, we introduce a new model and analyze it with respect to the criteria. The choice of complexity measure constitutes the major difference between the various asynchronous models proposed. In this model, the running time of an algorithm is defined to be the expected maximum number of steps taken by any processor, where the expectation is calculated given a distribution on all possible schedules and the number of steps can be normalized by a weighting function for relative processor speeds (as discussed in §7). Finally, in §§5 and 6 we analyze the paradigms of synchronization and pointer jumping, respectively. The results establish the complexities of various fundamental algorithms and the desirability of our complexity measure. The analyses themselves present general techniques for handling the dependencies that arise between processor actions.

2. Criteria for asynchronous models. Our first task is to provide a natural framework for considering algorithms performed by processors communicating asynchronously through shared memory cells. The removal of a global clock from a PRAM model leaves undefined several aspects of the resulting asynchronous model. On a PRAM, we can assume that a timestep is broken into separate read, local computation, and write phases, and that read and write conflicts are resolved by various rules. In an asynchronous environment, resolving read and write conflicts is less straightforward. In particular, a read and a write can occur simultaneously. From this possibility arises the question of which possible values may be returned by a read concurrent with a write. The choice of a *synchronization primitive*, discussed in §2.1, indicates which operations are atomic, and thus settles this concurrency question for our purposes. The choice of a synchronization primitive will induce a set of allowable schedules of processor steps, which may or may not be further restricted in the definition of the model. For the most general purposes, a model should make use of a weak synchronization primitive, and any algorithm on the model should perform correctly for any possible schedule of processor steps. Section 2.2 is concerned with the attributes of complexity measures. In §2.3, we identify three basic paradigms that serve as building blocks for many parallel algorithms. The quality of a complexity measure can be tested by applying it to these paradigms.

2.1. Synchronization primitives. The power of a particular model of computation depends heavily on the synchronization primitives that are assumed, as well as on assumptions about execution time. At one end of the spectrum is the synchronous PRAM model, which makes use of atomic synchronization barriers that cost unit time. The schedule of processor steps is constrained to allow each processor to take at most one step between each pair of barriers. It is the atomicity of the execution of barriers that makes the PRAM a synchronous model; by removing the assumption of a global clock, we are replacing this strong synchronization primitive with a weaker one. The choice of a primitive is a parameter of our model.

In this paper, we consider the atomic read/write cell [15], [16]. The reader is referred to Herlihy [11] for a discussion of several primitives and their relative strengths in an environment where processors are subject to arbitrary delays; under such a measure, the atomic read/write cell is the weakest of the primitives studied. Intuitively, reads and writes are the only atomic operations applied to such a cell; these operations are seen as covering intervals, not points, in time. An assignment of processor actions to time intervals is a *schedule*. Operations in a schedule are concurrent if and only if the associated time intervals overlap. The following property can be guaranteed of accesses to an atomic read/write cell: for any schedule, the values returned by the reads are consistent with some total ordering of processor steps formed by shrinking each time interval to a point within the interval. This property is also known as the correctness condition of *linearizability* [12]. When the accesses to each cell are linearizable, it can be shown that all actions taken together are linearizable. Herlihy and Wing [12] argue that linearizability is the basic correctness condition needed for reasoning about concurrent objects. In this sense, the atomic read/write cell is the weakest synchronization primitive.

2.2. Complexity measures. The absence of a global clock results in the absence of an obvious choice for a time complexity measure. Our complexity measure should reflect the actual elapsed time achieved on real machines, and therefore should appropriately capture the notions of processor speed and load balance. The relationship with processor speeds can be phrased in the following simple-minded fashion. Consider the time complexity, according to some measure, of executing an algorithm with processors operating at certain fixed speeds. If we then consider the complexity when several of the processors are replaced by faster (or slower) ones, the time measure should not increase (respectively, decrease). The issue of load balance can be illustrated by considering an execution in which all processors have the same speeds and the work is balanced among the processors. If we compare the complexity with that arising from a situation in which the work is unbalanced, it would be expected that the complexity should not be any higher in the balanced case. Intuitively, if the machine speeds are close and the parallelism is high, the time measure should be low; if imbalance in speeds results in the inefficient use of processors, the time measure could be high. Finally, we would like our measure to be a generalization of the measure used in the synchronous PRAM model, so that comparisons between synchronous and asynchronous complexities have some meaning.

It is important to note that the initial and final locations of information may be key factors in influencing an algorithm's complexity (or even whether or not a problem is tractable). In particular, it may be important to specify whether inputs are distributed in local memory or stored in shared memory. It may also be important to define at what point an algorithm has been completed, where completion conditions range from all processors knowing that the algorithm has been completed to having the correct answer somewhere in global memory. Unless otherwise stated, we will assume that initial values are held in global memory and that the algorithm has terminated when some processor writes into a special write-once answer cell. This implies that at least the writing processor is aware that the computation has been completed.

2.3. Paradigms. The behavior of various algorithms in an asynchronous setting depends on the degree of interaction between processors. In the following, we distinguish between *problems*, particular *algorithms* to solve problems, and *paradigms*, or classes of algorithms. The *independence* paradigm captures algorithms in which the processors work independently from one another. The synchronous cost of an algorithm in the independence paradigm is simply the maximum number of steps taken by any of the processors. Since in this case there is no interaction at all, it would seem that the synchronous and asynchronous time measures should be identical.

In contrast, the *synchronization* paradigm, encompassing such algorithms as the execution of a synchronization barrier and the computation of the OR of the inputs, involves strong interaction between processors. In essence, each processor must wait until all processors are ready to synchronize. Equivalently, an algorithm in this paradigm can be seen as delaying all processors until each processor has reached a particular step in its program. In a synchronous model, the execution of a synchronization barrier is assumed to be atomic; to compute the OR in an asynchronous model, the straightforward adaptation of the constant time synchronous algorithm does not yield a correct solution. In fact, it does not seem likely that a synchronization barrier can be executed in constant time in an asynchronous setting.

The third paradigm is *pointer jumping*, in which processors interact, but at the same time adapt to each others' speeds. In algorithms of this type, the inputs initially form a linked list in global memory and each processor holds in local memory the address of a unique node in this list. The particular algorithms will differ in the values held at each node and the functions used to update them. For example, in a list ranking algorithm, the value at a node at any time indicates the number of successors of the node detected thus far. The program executed by a particular processor consists of phases, where in each phase a processor chooses a node, reads its pointer, follows the pointer to read its successor's value and pointer, and then updates its node's value and pointer. As such an algorithm progresses, each node's pointer advances toward the tail of the list, coalescing information along the way, using its successor's pointer and value. When all pointers have reached the tail of the list, the algorithm has been completed. In a particular synchronous algorithm, each processor is assigned a particular node, which it repeatedly updates. This synchronous algorithm on n inputs consists of $\log n$ phases. After phase i each processor points to the cell that was 2^i links away in the original list or points to the tail of the list. Since each phase consists of $O(1)$ timesteps, the running time is in $O(\log n)$.

A single slow processor will not substantially alter the $O(\log n)$ running time of this algorithm. Each processor that reads the slow processor's cell has just one more pointer to read; when the slow processor gets around to taking a step, it benefits from the work done by the owner of the cell it reads. It seems that the difference between the synchronous and asynchronous complexity of pointer jumping should be greater than the difference between synchronous and asynchronous complexity of the independence paradigm and less than that of the synchronization paradigm.

3. A new model and measure.

3.1. Definition of the model. In this section, we introduce a new model which consists of p processors communicating asynchronously through m shared atomic read/write cells. As in the PRAM model, each processor is a random access machine executing a program made up of reads and writes to global memory and local operations [1]. The model is based on most PRAM assumptions: the existence of unique processor IDs, the lack of faults of any kind, and the equality of costs of all operations. However, in a departure from the PRAM, there is no assumption of synchrony and hence no implicit global clock. Consequently, although the steps of a particular processor's program are executed sequentially, global memory accesses by different processors can be interleaved or overlapped in arbitrary ways, resulting in arbitrary schedules of processor steps.

In the absence of a global clock, it is not immediately clear how time can be measured; it is this aspect that gives rise to the widest divergence among asynchronous models, as will be discussed in §3.4. A possible starting point is the synchronous complexity measure, which is usually stated in terms of the global clock. Removing the clock from discussion, the synchronous time complexity of an algorithm on a particular input can also be expressed as the maximum number of steps taken by any processor in the course of the execution. In the asynchronous setting, the actions performed in the execution of a particular algorithm on a particular input might depend on the particular schedule of processor steps that occurs. Consequently, the maximum number of steps taken by any processor might depend on which schedule occurs. A worst-case measure can be determined by maximizing over all schedules. If all possible schedules are allowable, then the worst-case measure can in essence be a determination of how well the algorithm degrades into a sequential algorithm. (Although this measure is of interest in its own right, it is based on an event that has very low probability, yielding a sequential measure.)

Instead, we consider time to be defined as the *expected* maximum number of steps taken by any processor, where the expectation is calculated given a distribution on all possible schedules. Instead of expressing this distribution in terms of schedules, we make use of the fact that the results of the accesses to all the memory cells in any schedule are as if the accesses occurred in a sequential order, as discussed in §2.1. We define a *linearization* for p processors to be a sequence of the processor IDs; a linearization represents a schedule for a particular computation if the result of the computation is as if steps were taken sequentially by the processors in the order specified in the linearization. Since there may be more than one total order of the actions consistent with a particular schedule, there may be more than one linearization that can represent a particular schedule. Each linearization represents at least one schedule, namely, a schedule in which no actions are concurrent. We can thus express a distribution on the set of all schedules by a distribution on the set of all linearizations, where if each schedule has a nonzero probability of occurring, each linearization has a nonzero probability of occurring.

In our formal definition of the time complexity of an algorithm, the numbers of steps taken by the various processors can be normalized by a weighting function for relative processor speeds, as discussed in §7. Here we define the time complexity of algorithm A when the weights of the p processors are all equal to one. For I an input of size n, δ a distribution on linearizations for p processors, and ℓ a linearization, the time complexity of A, $t_A(p, n)$, is

$$\max_I \sum_\ell Pr_\delta[\ell] \cdot \max_j \left\{ \begin{pmatrix} \text{number of steps taken by processor } j \text{ on input } I \\ \text{using algorithm } A \text{ with linearization } \ell \end{pmatrix} \right\}.$$

The time complexity of a problem can be defined, as in the synchronous setting, as the minimum over all algorithms solving the problem of the time complexity of the algorithm.

3.2. Intuition. To develop our intuition, we consider the pointer jumping paradigm, and determine the maximum number of steps taken by any processor for several specific linearizations. To begin, we focus on a simple situation in which processors progress at roughly the same rate. We will say that a linearization of p processors is a *round-robin ordering* if for all i, steps $pi + 1$ through $p(i + 1)$ of the linearization contain exactly one step by each processor. In the case of a round-robin ordering, the progress made is akin to that made in the synchronous environment, and the maximum number of steps taken by any processor is $O(\log p)$. Now suppose that we altered the above linearization by delaying one processor P. Any processor assigned to a cell closer to the tail of the list than P's cell would not be affected by P's actions. The progress by processors farther from the tail may be slowed by one doubling phase, or a constant number of steps, over their progress in the round-robin case. The processor P itself will take at most $O(\log p)$ steps, yielding an overall maximum of $O(\log p)$ steps. On the other hand, the degenerate sequential case (in which processor i takes all its steps before processor $i + 1$, where processors are numbered in increasing order from the head of the list) takes $O(p)$ time by our measure. This makes sense; it is very far from being synchronous and hence is not making good use of the parallelism.

3.3. Evaluation of the new model. Our model fulfills the criteria outlined in §2. The synchronization primitive assumed is a very weak one, the atomic read/write cell, the use of which can be simulated on models with stronger primitives. The model does not pose any restrictions on the class of allowable algorithms. Correct algorithms are correct for any possible schedule of processor steps. The complexity of an algorithm does not depend on restrictions on the set of schedules that may occur, only on the probability distribution on the schedules in the set. The use of a weighting function (§7) allows for the complexity measure to differentiate between replacing a subset of the processors by slower processors and by faster processors. Since the synchronous measure is in fact the expected maximum number of steps taken by any processor, the time complexity on our model, under the distribution in where there is a single possible interleaving, can be compared to the time complexity on synchronous PRAMs.

Finally, the complexities of the three basic paradigms for distributions in $\mathcal{C}$ and $\mathcal{S}$ (as defined in §4) conform to our intuition. Pointer jumping and synchronization both have complexity $O(\log n)$ for problems of size n using n processors; this will be discussed in §§4–6. In the independence paradigm, each processor exits the computation as soon as it has completed its individual steps. Thus, the number of steps taken by a processor is the same for all linearizations. The time required is exactly the maximum number of steps in any processor's program.

3.4. Related work. In the last few years, several different asynchronous models have been proposed; we restrict our attention to those which communicate through shared memory. These models differ from ours primarily in the definition of complexity measures. Each measure is defined to avoid the degenerate sequential case; some restrict the legal linearizations, some restrict the algorithms under consideration.

Gibbons [8], [9] observes that although any synchronous algorithm can be transformed into a synchronous one by inserting a synchronization barrier after each step, for certain algorithms fewer synchronization barriers are necessary. To keep analysis simple, Gibbons stipulates that if one processor reads a cell and another processor writes the same cell, there must be a synchronization barrier between these two actions. This results in the inclusion of barriers, and consequently the increase in running time, in algorithms for problems that have no inherent need for synchronization. Moreover, the set of allowable algorithms is constrained by this rule. The time complexity of a sequence of steps between synchronizations, a *phase*, is taken to be the maximum number of steps taken by any processor during the phase. The insertion of a synchronization barrier after each phase ensures that the algorithm is slowed to the speed of the slowest processor each phase. Although the complexity measure remains the same no matter what interleaving of processor steps occurs, the actual elapsed time is in fact very sensitive to changes in speeds of processors.

Valiant's [25], [26] message-passing Bulk-Synchronous Parallel model (BSP) shares with Gibbons' model the division of processing into phases between synchronization barriers. Relying on a sufficiently large number of virtual processors for which an algorithm is written in comparison to the number of processing units on the actual machine, a balance between computation and communication costs can be realized. Valiant argues that his model captures the true essence of parallel computation by bridging shared memory and message-processing models. As a consequence of that very generality, however, the BSP is not sufficiently fine-grained to allow for a full investigation of the power of asynchrony in a shared memory setting.

The model proposed by Kruskal, Rudolph, and Snir [14] and Cole and Zajicek [2] measures time as the maximum number of minimal rounds needed to complete the algorithm, where a *round* is a sequence of steps during which each processor takes at least one step (the notion of rounds was first introduced by Fischer and Lynch [6]). This has the effect of removing certain linearizations from consideration. In the case of pointer jumping, the maximum occurs in the synchronous case, when each processor takes only a single step each round.

Martel, Park, and Subramonian [18] consider an asynchronous model subject to fail-stop errors, where the schedule is chosen by an adversary. As ammunition to use against the adversary, the processors each have independent random number generators, used to perform probabilistic computations. Since all algorithms in this model are probabilistic, this model is best compared to a probabilistic PRAM model.

More recently, in independent work Cole and Zajicek [3] have defined a model based on distributions of interleavings. Their unbounded delay model consists of a class of distributions that is strictly contained in the classes of distributions considered in this work.

4. Probability distributions. It seems that our measure is only as good as the choice of a distribution over linearizations. Ideally, it is desirable to be able to analyze each algorithm with respect to all possible distributions. We begin by considering the behavior of asynchronous algorithms under certain distributions, chosen for their proximity to possible behavior of actual processors. These distributions are fairly close to the synchronous setting, which can be seen in this model as subject to a distribution on linearizations, all of which are round-robin orderings of processor IDs. In §§4.1 and 4.2, we define two general classes of distributions and state facts that apply to these classes. The proofs of the facts, which use only elementary probability theory, appear in [21]; Facts 4.5 and 4.9 are consequences of Raghavan's version of the Chernoff bound [22]. In §4.3, observations are made concerning the application of facts to algorithmic complexity and the relations of the distributions to those used in other work.

4.1. Simple delay distributions. A simple way of generating a distribution is by allotting steps in an approximate round-robin ordering, where a processor may miss a step from time to time. Such a distribution can be represented by a set of Markov chains, with one chain per processor. Transitions are made simultaneously on all Markov chains, and the resultant outputs are ordered to form a linearization. The only type of output symbol generated by the Markov chain for the ith processor is the ID i. The outputs are ordered according to a sequence of permutations of processor IDs, one permutation per transition; the outputs resulting from a particular transition are ordered in a manner consistent with the permutation for that transition. Recall that a linearization represents a schedule of possibly overlapping processor actions (§3.1); the simultaneous transitions of the Markov chains are used to create linearizations that approximate round-robin orderings and hence do not necessarily imply simultaneous processor actions. The simplest nontrivial distributions of this type are generated by Markov chains with two states, one corresponding to a step (an output state), the other to idle time (a nonoutput state); we call these the *simple delay* distributions. We define the time complexity of an algorithm A for a fixed set of two-state Markov chains. For I an input of size n, $\mathcal{O}$ a sequence of permutations of IDs of p processors, and ℓ a linearization for p processors, the complexity of A, $t_A(p, n)$, is

$$\max_I \max_{\mathcal{O}} \sum_{\ell} Pr_{\mathcal{O}}[\ell] \cdot \max_j \left\{ \begin{pmatrix} \text{number of steps taken by processor } j \text{ on input } I \\ \text{using algorithm } A \text{ with linearization } \ell \end{pmatrix} \right\}.$$

As before, the time complexity of a problem is defined analogously. Let ρ_i be the probability of transition from the step state to the step state in the Markov chain for processor i, and let $\bar{\rho}_i$ be the probability of transition from the idle state to the step state. In subsequent sections we will consider the set of distributions $\mathcal{S}$, defined as follows.

DEFINITION 1. *The set $\mathcal{S}$ is the set of simple delay distributions such that all ρ_i's and $\bar{\rho}_i$'s are within a constant factor of each other independent of p, and are in $\Omega(1/p^{O(1)})$.*

The Markov chain for one processor is illustrated in Fig. 1, for $\rho = 1/3$ and $\bar{\rho} = 1/4$.

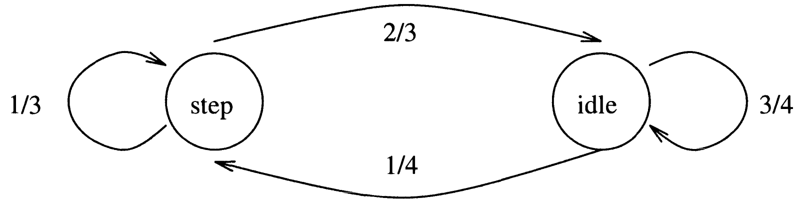


FIG. 1 *A simple delay distribution.*

Distributions of this type are intended to approximate slight deviations from a synchronous setting. In order to assess the accuracy of the approximation, we characterize the synchronous setting in terms of possible linearizations and their frequencies, and then compare this characterization with that of the simple delay distributions. Since in a synchronous PRAM a write never takes place at the same time as a read, each read or write phase can be seen as adhering to a round-robin ordering of the processor steps. If we slightly modify this model to allow writes to take place at the same time as reads, it is possible that in a round-robin ordering more progress could be made than if all actions occurred in lock-step. For example, whereas in the synchronous setting if one processor writes to a memory cell, another processor can read it only in the next read phase; here the write can precede the read in a round-robin ordering. We avoid this possibility by defining the complexity of A so that the steps within a transition occur in the least advantageous possible order. The simple delay distributions deviate further from the synchronous setting by allowing the possibility that in some phases not all processors take steps.

The next facts are of use when considering simple delay distributions. Here we relate the expected maximum number of steps taken by any processor to the number of transitions. Recall that for these distributions, the transitions for the processors are in lock-step, though the resulting set of processor steps may not be. Let $\sigma_* = \min_i\{\rho_i, \bar{\rho}_i\}$ and let $\sigma^* = \max_i\{\rho_i, \bar{\rho}_i\}$. By definition of $\mathcal{S}$, $\sigma_* \in \Theta(\sigma^*)$ and $\sigma_* \in \Omega(1/n^{O(1)})$.

FACT 4.1. *The expected maximum number of steps taken by any of n processors in an expected t transitions is in $O(t\sigma^*)$, for $t \in \Omega((\log n)/\sigma^*)$.*

FACT 4.2. *For any $g \leq n$, the expected number of transitions needed to allot at least one step to each of a fixed set of g out of n processors is in $\Omega((\log g)/\sigma^*)$, provided that $\sigma^* < k/\ln g$ for a constant $k < 1$.*

FACT 4.3. *The expected number of transitions needed for each of n processors to take at least $kt\sigma_*$ steps is in $\Theta(t)$ for any constant $k > 0$ and for $t \in \Omega((\log n)/\sigma^*)$.*

FACT 4.4. *Suppose that t transitions, $t \in \Omega((\log n)/\sigma_*)$, are taken by n processors. For $t = (c' \ln n)/\sigma_*$, there exists a constant $c > 0$ depending on c' such that the probability that there is a processor that takes fewer than $ct\sigma_*$ steps is at most $1/n^{O(1)}$.*

FACT 4.5. *If the expected number of steps taken by a processor is at least E, then the actual number of steps taken by that processor is at least $E/2$ with probability at least $1 - e^{-E/8}$.*

By using Fact 4.1 in upper bounds, we can determine the expected maximum number of steps per processor by simply finding the total number of transitions taken. If we do not know the total number of transitions, it in fact suffices to know the expected total number of transitions. Facts 4.2 and 4.3 have the following useful consequence. Suppose we take enough steps to ensure that each processor takes at least one step. By taking a constant factor more steps, we are ensured that each processor takes at least $\log n$ steps.

4.2. Constant speed distributions. We may instead wish to model a situation in which processor actions are not independent of each other, such as when one processor's use of the network slows down another processor's accesses. We can assign to each processor j a probability ρ_j of being allotted the next step in the linearization, where $\sum_{j=1}^{p} \rho_j = 1$. We call such a distribution a *fixed speed* distribution, since it corresponds to processors maintaining particular speeds throughout the course of the computation. A fixed speed distribution can be represented as the vector $(\rho_1, \ldots \rho_p)$. Each ρ_i is a function only of p, the total number number of processors. A *uniform distribution* is a special case of a fixed speed distribution, where $\rho_i = 1/p$ for all i. We consider the following set of distributions.

DEFINITION 2. *The set $\mathcal{C}$, or* constant speed *distributions, is the set of fixed speed distributions where for all i and j, ρ_i/ρ_j is bounded by a constant independent of p.*

In the simple delay distributions, each transition by the various Markov chains could be seen as generating a section of the linearization that approximated a round-robin ordering. In constant speed distributions, however, there is no local sense in which the linearizations produced approximate those associated with the synchronous setting. However, the deviations remain moderate in that in a long enough section of a linearation, with high probability each processor will take a number of steps roughly proportional to its speed.

The following facts allow us to determine the expected maximum number of steps taken by any processor by first determining the expected total number of steps taken by all processors. Throughout this section, it is assumed that all processors are active until the end of the computation. In Facts 4.6–4.9, we consider an arbitrary distribution δ on linearizations of n processors such that $\delta \in \mathcal{C}$. Let $\gamma^* = \max_i\{\rho_i\}$ and let $\gamma_* = \min_i\{\rho_i\}$. We know that γ^* and γ_* are within a constant factor of each other, by the definition of $\mathcal{C}$.

FACT 4.6. *The expected maximum number of steps taken by any of the processors in a linearization of expected length t is in $O(t/n)$, provided that $t \in \Omega(n \log n)$.*

FACT 4.7. *For any $g \leq n$, the expected number of steps needed to guarantee that each of a fixed set of g out of n processors take at least one step is in $\Theta(n \log g)$.*

FACT 4.8. *The expected number of steps needed to guarantee that each processor takes at least $kt\gamma_*$ steps is in $\Theta(kt)$, for any constant $k > 0$ and for $t \in \Omega((\log n)/\gamma_*)$.*

FACT 4.9. *If the expected number of steps taken by a processor is at least E, then the actual number of steps taken by that processor is at least $E/2$ with probability at least $1 - e^{-E/8}$.*

Facts 4.6–4.9 are analogous to Facts 4.1, 4.2, 4.3, and 4.5, respectively, and have the same uses as noted in §4.2. In particular, as a consequence of Facts 4.2 and 4.3, the number of transitions needed to obtain $O(\log n)$ steps per processor is within a constant factor of the number needed to ensure one step per processor. Note, in Fact 4.6, that the expected total number of steps divided by the number of processors will be equal to the time, as in the synchronous case.

4.3. Further observations about distributions.

4.3.1. Other useful theorems. In each algorithm considered in this paper, it is assumed for simplicity that all processors remain in the computation until its completion. Since allowing processors to exit early will only decrease the complexity of an algorithm, the results are not restricted by this assumption. However, in the case of lower bounds, the issue of early exiting processors cannot be ignored.

The proof of Theorem 4.10 follows from the facts in §§4.1 and 4.2; it can be found in [21].

THEOREM 4.10. *Let X be an algorithm such that no processor can exit the computation until all n processors have entered the computation. Then the algorithm X requires $\Omega(\log n)$ time using n processors for distribution δ on linearizations such that $\delta \in \mathcal{C} \cup \{s \in \mathcal{S} \mid \sigma^* < k/\ln n$ for a constant $k < 1\}$.*

THEOREM 4.11. *If an algorithm A can be decomposed into A' and A'' such that the preconditions for A'' are a subset of the postconditions for A', then the time needed to execute A is at most the sum of the times needed to execute A' and A''.*

4.3.2. Distributions and other models. Another way to view distributions on linearizations is by considering the distribution on times taken between steps by each processor. The uniform distribution corresponds to choosing this waiting time for each processor according to an exponential distribution, independent of all other processors.

Many other models can be characterized as special cases of the general model, with specific distributions and restrictions on algorithms imposed. A special case of a simple delay distribution, where $\rho_i = \bar{\rho}_i = \rho$ for all i and ρ is bounded below by a constant, yields Cole and Zajicek's [3] unbounded delay model. The synchronous PRAMs are generated by distributions in which only linearizations corresponding to round-robin orderings have nonzero probabilities; the only algorithms allowed are those where reads and writes occur in alternating rounds. In the case of the PRIORITY PRAM, steps $pi + 1$ to $p(i + 1)$ must be ordered from lowest priority to highest. The various types of global access result from further restrictions on allowable algorithms. Probabilistic models of fail-stop errors, such as those considered by Kedem, Palem, and Spirakis [13], are formed by simple delay distributions where $\rho_i = \rho$ and $\bar{\rho}_i = 1$ for all i, so that once a step is missed, the processor takes no further steps.

It is important to note that the constant speed distributions differ from any of the distributions considered by Cole and Zajicek [3]; to see this, consider in the uniform distribution the probability that a processor takes three consecutive steps without any steps by other processors intervening. In essence, the uniform distribution provides a more pessimistic view of the world, since greater deviations from a round-robin allocation of steps are more likely to occur. This pessimism results in more robust upper bounds.

5. Barrier synchronization.

In this section we establish the upper and lower bounds of $\Theta(\log n)$ on the complexity of executing a synchronization barrier by n processors. We then show how this result can be used to obtain complexities of various algorithms, including general classes of algorithms transformed in a uniform way from their synchronous counterparts.

5.1. Time complexity analysis.

A synchronization barrier can be seen as an algorithm that begins by each processor signaling that it is ready to synchronize and ends by each processor knowing that all the other processors have signaled. In particular, no processor can exit the computation until all processors have entered the computation. It is in fact possible to obtain a tight bound on the complexity of synchronization.

THEOREM 5.1. *Synchronization of n processors can be performed in $O(\log n)$ time for any distribution δ on linearizations such that $\delta \in \mathcal{C} \cup \mathcal{S}$. Synchronization of n processors requires $\Omega(\log n)$ time for any distribution δ' on linearizations such that $\delta' \in \mathcal{C} \cup \{s \in \mathcal{S} \mid \sigma^* < k/\ln n$ for a constant $k < 1\}$.*

Proof. We let $\sigma^* = \max_i\{\rho_i, \bar{\rho}_i\}$ in the simple delay distributions, and we note in square brackets the modifications needed to alter the proof for constant speed distributions to one for simple delay distributions.

The lower bound is a consequence of Theorem 4.10. For the upper bound, we consider an algorithm operating on a binary tree all of whose nodes initially contain the value 0 (it is not clear how the algorithm can work without some assumption on initialization). We will say that a node in the tree has been *filled* if some processor has written a 1 in it. Each processor begins by filling a leaf node to indicate that it is ready to synchronize. When the root node has been filled, all processors are ready to synchronize; a particular processor completes its computation of the synchronization as soon as it reads a 1 from the root node. The reading of the root can be accomplished in one more step by each processor, or in $O(\log n)$ time, by Facts 4.7 and 4.6 [by Facts 4.2 and 4.1] and by having processors alternate between steps taken to do work in the tree and steps taken to read the root node. It will suffice to analyze the filling of the nodes of the tree.

The nodes of the tree are filled from the bottom up, with each processor attempting to fill one node at each level. The nodes that a processor attempts to fill form a path from a leaf to the root. A node can be filled only after both its children have been filled. More formally, we make the following definitions. A distinct leaf node, ℓ_P, is assigned to each processor P. The nodes on the path from ℓ_P to the root of the tree form the set nodes(P). For a node u, let proc(u) be the set of processors P such that $u \in$ nodes(P). The set proc(u) is exactly the set of processors that attempt to fill u. Equivalently, proc(u) can be seen as the set of processors P such that ℓ_P is a leaf in the subtree rooted at u.

The processors will have completed the algorithm as soon as the root is filled. Suppose that each processor takes $\Theta(\log n)$ steps. By Fact 4.8 [Fact 4.3], the expected total number of steps will be in $\Theta(n \log n)$ [the expected number of transitions will be in $\Theta(\log n/\sigma^*)$]; by Fact 4.6 [Fact 4.1], the expected maximum number of steps per processor will be in $O(\log n)$.

To complete the proof of the theorem, it will suffice to show that if each processor takes $\Theta(\log n)$ steps, the root will be filled, regardless of the ordering of steps. The key notion is that of delay.

DEFINITION 5.2. *A processor P is* delayed at node u *if the child of u in* nodes(P) *has been filled, the other child of u has not been filled, and it is P's turn to take a step.*

DEFINITION 5.3. *A processor P is* delayed in the subtree rooted at u *if $P \in$* proc(u) *and P has been delayed at a node $v \in$* nodes(P), *where v is a descendant of u (note that v can be u itself). Let* delay(u) *be the set of processors delayed in the subtree rooted at u.*

The lemma below shows that at least one processor is never delayed in the tree; such a processor can fill the root in $\Theta(\log n)$ steps. This completes the proof of the theorem.

LEMMA 5.4. *There exists at least one processor $P \in$* proc(u) *such that P is not delayed in the subtree rooted at u; equivalently,*

$$|\text{delay}(u)| < |\text{proc}(u)|.$$

Proof. We prove our lemma by induction on the height of u. The base case is trivial: At a leaf, no processor is delayed. Now consider a node u, with children x and y. Note that $\text{proc}(u) = \text{proc}(x) \cup \text{proc}(y)$.

Without loss of generality, assume that x is filled before y. Consider a processor $P \in \text{proc}(y)$; it is clear that P cannot be delayed at u. However, for all $Q \in \text{proc}(x)$ it is possible for Q to be delayed at u, in particular if Q takes a step after x has been filled but before y is filled. We then determine that the number of processors delayed in the subtree rooted at u is

$$|\text{delay}(u)| \leq |\text{proc}(x)| + |\text{delay}(y)|.$$

By the induction hypothesis, $|\text{delay}(y)| < |\text{proc}(y)|$. Thus $|\text{delay}(u)| < |\text{proc}(u)|$, as needed. □

5.2. Consequences.

5.2.1. Processor-efficient solutions. With a bit of refinement to the algorithm presented above, the synchronization paradigm can be used to obtain processor-efficient algorithms for various problems. Although a synchronization problem of size n, by definition, requires n processors, other problems in the synchronization paradigm can be solved using $n/\log n$ processors. Note, furthermore, that the lower bound of Theorem 4.10 does not apply to algorithms in which certain processors may exit before each processor has taken a step. Whether or not there is an $\Omega(\log n)$ lower bound for OR in such a circumstance remains an open question.

THEOREM 5.5. *The n-input problems of* OR, AND, MAX, MIN, *and* PARITY *can be solved in $O(\log n)$ time using $n/\log n$ processors for any distribution δ on linearizations such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

Proof. The algorithms for the various problems are very similar; here we concentrate on the algorithm that computes the OR of n bits. We can think of the input as being partitioned into $n/\log n$ blocks of $\log n$ bits each. In the first stage, each processor reads a different block of bits and computes the OR of these $\log n$ bits sequentially, in time in $O(\log n)$.

In the second stage, the processors perform an algorithm similar to that described in Theorem 5.1. We make use of a binary tree with $n/\log n$ leaves. Each leaf is filled with the OR of $\log n$ bits (as calculated in the first stage); each interior node is filled with the OR of the values of its children. By Theorem 5.1, the time complexity of this algorithm is in $O(\log(n/\log n)) = O(\log n)$. Since the time required for each stage is in $O(\log n)$, the total running time is in $O(\log n)$, as claimed. □

5.2.2. Simulation of circuits and asynchronous PRAMs. Since a Boolean formula, and hence a Boolean circuit, can be viewed as a tree, it is not difficult to transform one into an asynchronous algorithm. For a sufficient number of processors, the depth of the circuit yields an immediate upper bound on the time of the algorithm, as in the following theorems.

THEOREM 5.6. *The problem of multiplying two $n \times n$ Boolean matrices can be solved in $O(\log n)$ time using $O(n^3/\log n)$ processors for any distribution δ on linearizations such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

Proof. Two n by n Boolean matrices can be multiplied by using a simple circuit of ANDs and ORs. There are $n/\log n$ processors assigned to each of the n^2 matrix positions. Each processor computes the AND of $\log n$ pairs of bits and then the OR of the $\log n$ results. The OR of the $n/\log n$ values for a particular position are then computed in time $O(\log n)$, yielding a total time complexity of $O(\log n)$. □

COROLLARY 5.7. *Transitive closure can be solved asynchronously in $O(\log^2 n)$ time using $O(n^3/\log n)$ processors for any distribution δ on linearizations such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

Proof. The transitive closure of a matrix, can be computed by taking the nth power of the matrix by $O(\log n)$ squarings, or a total running time of $O(\log^2 n)$. □

Further results can be obtained by simulating algorithms developed for Gibbons' ASYNCHRONOUS PRAM. Since the algorithms considered by Gibbons [8], [9] alternate between the independence paradigm and the synchronization paradigm, we can easily simulate these algorithms and analyze their running time on our model. The complexity results obtained on an ASYNCHRONOUS PRAM with a charge for latency can be obtained (or improved) on an ASYNCHRONOUS PRAM where there is no charge for latency. The following theorem is a direct consequence of the construction of the ASYNCHRONOUS PRAM algorithms and Theorem 4.11; the function $B(n)$ is the cost of executing a synchronization barrier on n items in Gibbons' model.

THEOREM 5.8. *Any problem that can be solved in $f(B(n))$ time on an n-processor* ASYNCHRONOUS PRAM *with no charge for latency can be solved on our model in $O(f(\log n))$ time using n processors for any distribution δ such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

COROLLARY 5.9. *The n-input problem of fast Fourier transform can be solved in time $O(\log^2 n/\log\log n)$ using $n\log\log n/\log n$ processors for any distribution δ such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

COROLLARY 5.10. *The problem of merging two bitonic lists can be solved in time $O(\log^2 n/\log\log n)$ using $n/\log n$ processors for any distribution δ such that $\delta \in \mathcal{C} \cup \mathcal{S}$.*

5.2.3. Simulation of synchronous PRAMs. As a consequence of Theorem 5.1, we obtain the following relation between synchronous and asynchronous complexity classes.

THEOREM 5.11. *Any problem that can be solved in t time on an* ARBITRARY CRCW PRAM *using p processors can be solved asynchronously in time $O(t(p/p' + \log p'))$ using p' processors for any distribution δ such that $\delta \in \mathcal{C} \cup \mathcal{S}$, and for $1 \le p' \le p$.*

Proof. It will suffice to show how a single write phase of ARBITRARY can be simulated by p' asynchronous processors in $O(p/p' + \log p')$ time. The read phase is analogous.

Each asynchronous processor is assigned $\lfloor p/p' \rfloor$ ARBITRARY CRCW PRAM processors. At a given PRAM step, each asynchronous processor executes the appropriate step taken by each of the $\lfloor p/p' \rfloor$ simulated processors. The p' processors then synchronize in $O(\log p')$ time using separate parts of memory. It is not difficult to see that the memory cell contents in simulating cells at the end of the synchronization will correspond to those obtained after a synchronous step. □

When $p' = 1$, the result corresponds to a single processor sequentially executing all $O(tp)$ work. The other extreme, when $p' = p$, corresponds to inserting a synchronization barrier after each PRAM step and the work decreases to $O(t\log p)$. When $p' \in O(p\log\log p/\log p)$, we obtain the result that any problem that can be solved in time t on an ARBITRARY CRCW PRAM using p processors can also be solved asynchronously in time $O(tp/p')$ using p' processors for any distribution δ such that $\delta \in \mathcal{C} \cup \mathcal{S}$, using within a constant factor the same total amount of work as in the synchronous case.

6. Pointer jumping. In the paradigm of pointer jumping, processors interact, but at the same time adapt to each other's speeds. This adaptation allows for an $O(\log n)$ time asynchronous pointer jumping algorithm, matching the synchronous time bound. Since pointer jumping forms the basis of many fundamental problems, it is no surprise that Theorems 6.3 and 6.13 yield identical complexity measures for other problems such as list ranking and suffix sum. By adding a preprocessing stage during which the list is made doubly-linked, we can compute for each node in an input tree such information as preorder number, post-order number, level, and number of descendants in $O(\log n)$ time using the Euler tour technique [24].

In the following treatment, we consider a particular pointer jumping algorithm that starts with n inputs forming a linked list in global memory. Each node is associated with a distinct processor which has its address stored in local memory. The program of each processor is identical to that in the standard synchronous algorithm; however, the set of tasks executed may vary with the linearization. As the algorithm progresses, each processor advances the pointer of its node toward the tail of the list. Repeatedly, it follows its node's pointer to read the successor's pointer, and then updates its node's pointer. The algorithm ends when all pointers point to the tail of the list. Alternatively, we could have all processors execute a synchronization barrier after updating their pointers to point to the end of the list. We show in Theorems 6.3 and 6.13 that the complexity of pointer jumping is in $O(\log n)$, and hence an extra $O(\log n)$ for synchronization does not change the overall complexity.

6.1. Time complexity analysis.

6.1.1. Simple delay distributions. We begin by analyzing the complexity of the pointer jumping algorithm when the linearization is chosen subject to a distribution in $\mathcal{S}$. The analysis for distributions in $\mathcal{C}$, which follows in §6.1.2, is more complicated.

The analysis of the algorithm is based on a very simple idea. We would like to divide the sequence of steps taken during an execution of the algorithm into $\log n$ phases, and show that after phase i, each processor's node is either pointing at least 2^i positions closer to the tail of the list than at the start of the computation, or pointing to the tail of the list. We will say that such a processor is *on-time at the end of phase i*, or more succinctly, *on-time at i*. If this were true, then after phase i any node would be at least as far along the list as the corresponding node after the ith step of a synchronous computation. After $\log n$ phases, each node would point to the tail of the list. Ensuring that each processor is on-time at the end of each phase, however, would require that a phase have a prohibitively large number of steps.

Instead, we look at phases of $\lceil 18/\sigma_* \rceil$ transitions and relax the requirement for progress (recall that σ_* is the minimum probability of any processor taking a step at any transition). Namely, we wish to show that with probability at least a constant, say $1/3$, an arbitrary processor P is on-time at the end of phase i. We divide the linearization into two stages, and show that in each stage the expected maximum number of steps taken by any processor is in $O(\log n)$. The first stage consists of $\log n$ phases; we show in Lemma 6.1 that at the end of this stage, each processor is pointing to the tail of the list with probability at least $1/3$. The second stage consists of the remainder of the algorithm. In Lemma 6.2, we show that this stage can also be completed in $O(\log n)$ time, yielding an overall time complexity of $O(\log n)$. We now consider the analysis of the stages one at a time.

LEMMA 6.1. *After the first* $\lceil 18 \log n/\sigma_* \rceil$ *transitions in a linearization chosen from a distribution in* $\mathcal{S}$, *for each processor* P, *the probability that* P *is pointing to the tail of the list is at least* $1/3$.

Proof. Since any processor P will point to the tail of the list if it is on-time at the end of the $(\log n)$th phase, it will suffice to prove by induction on phase numbers that for phase i, P is on-time at i with probability at least $1/3$.

A processor is on-time at the end of phase i if there were at least two times during phase i at which it updated its node's pointer to point to the node of a processor that was on-time at the end of phase $i-1$. For convenience, we will use the phrase "*P updated to a cell of a processor on-time at $i-1$*" to express the notion that P updated its node's pointer to point to a node of a processor that was on-time at the end of phase $i-1$. We have a lower bound on the probability that an arbitrary processor P is on-time at i, that is, the probability that P updated to cells of at least two processors that were on-time at $i-1$.

For any events A and B, we know that $Pr[A] \geq Pr[A|B] \cdot Pr[B]$. Here we let A be the event that P updated to cells of at least two processors that were on-time at $i-1$ and B be the event that P took at least 9 steps. It will now suffice to determine lower bounds on $Pr[A|B]$ and $Pr[B]$.

We first determine a lower bound on the probability that P took at least 9 steps in a particular phase. Since P takes a step with probability at least σ_*, it is clear that $E \geq 18$, and hence by Fact 4.5

$$Pr[B] \geq 1 - e^{-2.25} \geq .89.$$

In order to show that $Pr[A] \geq 1/3$, it will now suffice to show that $Pr[A|B] \geq .38$. Of the 9 or more steps taken by P during phase i, there will be at least four reads followed by writes, updates to nodes of four different processors. By the induction hypothesis, each of these processors was on-time at the end of phase $i-1$ with probability at least $1/3$. Relying on the fact that in this type of distribution the probability of each processor taking steps is independent of the steps taken by any other processor,

$$\begin{aligned} Pr[A|B] &\geq 1 - Pr[\text{none are on-time}] - Pr[\text{exactly one is on-time}] \\ &\geq 1 - (\tfrac{2}{3})^4 - 4 \cdot \tfrac{1}{3} \cdot (\tfrac{2}{3})^3 > 0.38, \end{aligned}$$

completing the proof of Lemma 6.1. □

We are now ready to determine the complexity of stage 2.

LEMMA 6.2. *Suppose that for each processor* P, *the probability that* P *is pointing to the tail of the list is at least* $1/3$. *Consider a linearization chosen according to a distribution in* $\mathcal{S}$. *Then the expected maximum number of steps taken by any processor in the second stage is in* $O(\log n)$.

Proof. Let a' be a constant defined later, and let an experiment consist of $a' \log n/\sigma_*$ transitions. For $t = a' \log n/\sigma_*$, we know from Fact 4.4 that there exists a constant a such that the probability that there is a processor that takes fewer than $a \log n$ steps is at most $1/n^{O(1)}$. We say an experiment is successful if each processor takes at least $a \log n$ steps and each processor reaches the end of the list. The expected number of experiments is at most $1/p$, where p is the probability of success. To obtain an upper bound of $O((\log n)/\sigma_*)$ on the expected number of transitions (to which we can apply Fact 4.1 to obtain our result), it will suffice to obtain as an upper bound on the probability of failure a constant less than 1.

Let $\mathcal{A}$ be the event that there exists a processor that takes fewer than $a \log n$ steps, and let $\mathcal{B}$ be the event that there exists a processor that does not reach the end of the list. Then

$$\begin{aligned} Pr[\text{Failure}] &= Pr[\mathcal{A} \vee \mathcal{B}] \\ &\leq Pr[\mathcal{A}] + Pr[\mathcal{B}|\neg\mathcal{A}]. \end{aligned} \tag{1}$$

The upper bound of $1/n^{O(1)}$ on $Pr[\mathcal{A}]$ is a consequence of Fact 4.4. To obtain a similar upper bound on $Pr[\mathcal{B}|\neg\mathcal{A}]$, we note that if a processor reads the node of a single processor whose node is pointing to the tail of the list, it too advances its pointer to point to the tail of the list. The probability of a successor pointing to the end of the list at any point in this stage is at least 1/3, and hence $Pr[\mathcal{B}|\neg\mathcal{A}] \leq (2/3)^{a \log n}$ which is bounded above by $1/n^{O(1)}$ for a suitable value of a. We can choose a' appropriately. □

We can now prove Theorem 6.3. By Fact 4.1, the expected maximum number of steps taken by any processor in stage 1 is in $O(\log n)$. By Lemmas 6.1 and 6.2, stage 2 can be completed in $O(\log n)$, yielding a total time of $O(\log n)$.

THEOREM 6.3. *Pointer jumping can be performed in time $O(\log n)$ using n processors for any distribution $\delta \in \mathcal{S}$.*

6.1.2. Constant speed distributions. The time complexity analysis for constant speed distributions is complicated by the fact that the probability of one processor taking a step is not independent of the steps taken by other processors. In this section we develop a general technique for handling such dependencies, which may prove of use in complexity analysis of other algorithms.

To determine the complexity of the algorithm given a linearization chosen according to a distribution in $\mathcal{C}$, we again divide the linearization into phases, this time of $(8w + 4)(c_*)^{-2}n$ steps, where c_* and c^* are constants such that $\gamma_* \geq c_*/n$, $\gamma^* \leq c^*/n$, and $w \geq 2$ is a value such that

$$\left(1 - \frac{2c^*}{c_*(2w+1)}\right)\left(1 - \left(\frac{2}{3}\right)^{w} - w \cdot \frac{1}{3} \cdot \left(\frac{2}{3}\right)^{w-1}\right) \geq 0.53.$$

(Note that each of the two factors increases as a function of w, so such a w exists for all values of c_* and c^*.) The complexity analysis for this type of distribution is complicated by the correlation between different processors being allocated steps. In a fixed total number of steps, if one processor takes a large number of steps, any other processor is less likely to take a large number of steps. In particular, suppose that processor P' reads the node of processor P during phase i. The fact that P' reads the node of P implies that a certain amount of progress, and hence a certain number of steps, were allocated to P' and the processors with nodes between the nodes of P' and P. Since these steps were allocated in this way, we know that they were not allocated to P and its successors. This decreases the probability that P and its successors were allocated steps, and hence the probability that P is on-time.

We consider the probability of P being on-time conditioned on progress made by its predecessors. We can represent the state of the algorithm at any time by a directed graph, as follows: each processor is represented as a vertex, and a pointer from the node of P' to the node of P in the list is represented as a solid edge from the vertex for P' to the vertex for P. Clearly each vertex will have a single solid outedge. In addition, a processor $\hat{P}$ may have read and calculated a new pointer value, for example to $\bar{P}$, but not yet written the new value from its local memory to update its pointer in global memory; the new pointer value is represented as a dashed directed edge from the vertex for $\hat{P}$ to the vertex for $\bar{P}$. The *configuration* of predecessors of P at a particular time is simply the induced subgraph on all vertices that can reach P by any combination of solid and dashed edges; the set of predecessors is defined not to include P. We let $Q^i{}_P$ be a configuration of the predecessors of P at the end of phase i. If we know the configurations Q_P^{i-1} and $Q^i{}_P$, we have partial information about the progress made during phase i by the predecessors of P at the end of phase i.

Consider a particular configuration at the end of phase $i - 1$. Fixing a configuration at the end of phase i induces a distribution on the number of steps that are allotted during phase i to the predecessors of P at phase $i-1$ and the number that are allotted to the nonpredecessors. Given an allotment of steps to the nonpredecessors, we would like to be able to argue that, with reasonable probability, P is allotted a certain number of steps. This requires that the set of nonpredecessors be fairly large. Notice that if a node is originally closer to the tail of the list than P's node, then its processor is never a predecessor of P.

DEFINITION 6.4. *A processor P is a* tail *processor if it is initially assigned one of the $\sqrt{n}-1$ nodes closest to the tail of the list; otherwise, P is a* head *processor.*

Let S_P be the set consisting of P along with the tail processors. Whenever P is a head processor, we can conclude that S_P contains no proper predecessors of P at any time.

Initially we concentrate only on the progress of the head processors; we modify the definition of on-time for such processors.

DEFINITION 6.5. *A head processor P is* on-time *at the end of phase i if it is pointing either to the node of a tail processor or to a node that was originally at least 2^i positions closer to the tail of the list.*

For convenience, let $C(P, i, Q)$ denote the condition that for a head processor P, and a configuration Q of the predecessors of P at i, $Pr[P$ is on-time at $i \mid Q] \geq 1/3$. Of particular interest is the condition $C(P, \log n, Q)$, since a processor that is on-time after the $(\log n)$th phase is pointing to a tail processor. We will say that a head processor pointing to a tail processor is *promoted.* Let $D(P)$ be the condition that after phase $\log n$, $Pr[P$ is promoted$] \geq 1/3$.

We divide the linearization into four stages, treating head and tail processors separately. We show that in each stage the expected maximum number of steps taken by any processor is in $O(\log n)$. As in §6.1.1, the division is part of the analysis, not part of the algorithm. In the first two stages, we consider the progress of the head processors. The first stage consists of $\log n$ phases; Lemma 6.11 below will show that with high probability, condition $D(P)$ will hold for each head processor at the end of stage 1. Stage 2 begins when the first $(8w+4)(c_*)^{-2}n \log n$ steps have been completed, and ends as soon as each head processor is promoted. The next stage ends when each tail processor has advanced its pointer to the tail of the list. Finally, in stage 4 each head processor takes at most one more step to point to the tail of the list and complete the pointer jumping problem. We now consider the analysis of the stages one at a time.

For stage 1, we would like to claim that $D(P)$ holds for each head processor P. Since any head processor will be promoted if it is on-time at the end of the $(\log n)$th phase, it would suffice to prove for each phase i and each configuration Q of predecessors of P at i that condition $C(P, i, Q)$ holds. However, for some pairs of configurations at phases $i-1$ and i, the probability of a processor being on-time may be much smaller than $1/3$. We instead show in Lemma 6.8 that with high probability $C(P, i, Q)$ holds. Consider a configuration ${Q^i}_P$ of the predecessors of P at i; associated with each linearization that leads to ${Q^i}_P$ is a particular configuration of the predecessors of P at $i-1$. We will say that a configuration Q_P^{i-1} is *good with respect to ${Q^i}_P$* if

$$Pr[S_P \text{ gets} > (4w+2)|S_P|/c_* \text{ steps during } i \mid Q_P^{i-1} \wedge {Q^i}_P] \geq 1 - e^{-\sqrt{n}}.$$

We will establish the fact that for a fixed configuration Q of predecessors of P at i, it is likely that the configuration at time $i-1$ was good with respect to Q. We first show in Lemma 6.6 that with very high probability, S_P will get more than $(4w+2)|S_P|/c_*$ steps during i. Then, we show that the configuration at time $i-1$ is good with respect to Q with very high probability, by a simple averaging argument in Lemma 6.7.

LEMMA 6.6. *For any configuration Q of the predecessors of P at i,*

$$Pr[S_P \textit{ gets} > (4w+2)|S_P|/c_* \textit{ steps during } i|Q] \geq 1 - e^{-2.5\sqrt{n}}.$$

Proof. Since the number of steps taken during a phase is fixed, there is a fixed number of possible sequences of steps taken during a phase and a probability associated with each one.

To determine the probability that S_P gets more than $(4w+2)|S_P|/c_*$ steps, we make use of the fact that the expected number of steps allotted to S_P in a phase of $(8w+4)(c_*)^{-2}n$ steps is at least $(8w+4)|S_P|/c_*$. From Fact 4.9 it follows that

$$Pr[S_P \text{ gets} > (4w+2)|S_P|/c_* \text{ steps}] \geq 1 - e^{-(8w+4)\cdot|S_P|/8c_*}.$$

By definition $w \geq 2$, $c_* \leq 1$, and $|S_P| = \sqrt{n}$, and hence

$$Pr[S_P \text{ gets} > (4w+2)|S_P|/c_* \text{ steps}] \geq 1 - e^{-2.5\sqrt{n}}. \qquad \square$$

We fix a configuration Q of predecessors of P at i and consider the possible previous configurations at $i-1$, which we call $Q_1, Q_2, \ldots$. Let $\mathcal{A}_j$ be the set of linearizations that result in Q at i given Q_j at $i-1$. Let $\mathcal{M}_j$ be the subset of $\mathcal{A}_j$ in which S_P gets at most $(4w+2)|S_P|/c_*$ steps during the phase i. It follows from the statement of Lemma 6.6 that

$$\sum |\mathcal{M}_j| \leq e^{-2.5\sqrt{n}} \sum |\mathcal{A}_j|.$$

LEMMA 6.7. *The configuration at $i-1$ is good with respect to Q with probability at least $1 - e^{-\sqrt{n}}$.*

Proof. Let $\mathcal{B} = \{j \mid |\mathcal{M}_j| \geq e^{-\sqrt{n}}|\mathcal{A}_j|\}$. Then the statement of Lemma 6.7 is

$$\sum_{j\in\mathcal{B}} |\mathcal{A}_j| \leq e^{-\sqrt{n}} \sum |\mathcal{A}_j|.$$

It follows from Lemma 6.6 and the definition of $\mathcal{B}$ that

$$\begin{aligned} e^{-2.5\sqrt{n}} \sum |\mathcal{A}_j| &\geq \sum |\mathcal{M}_j| \geq \sum_{j\in\mathcal{B}} |\mathcal{M}_j| \\ &\geq \sum_{j\in\mathcal{B}} e^{-\sqrt{n}}|\mathcal{A}_j| = e^{-\sqrt{n}} \sum_{j\in\mathcal{B}} |\mathcal{A}_j| \end{aligned} \tag{2}$$

which implies the desired result. $\square$

LEMMA 6.8. *For any phase i, head processor P, and configuration Q of predecessors of P at i, $C(P, i, Q)$ is true with probability at least $1 - (w+1)^i e^{-\sqrt{n}}$.*

Proof. We prove this lemma by induction on the phase number. A processor is on-time at the end of phase i if there were at least two times during phase i at which it updated its node's pointer to point to the node of a processor that was on-time at the end of phase $i-1$. Consequently, we have a lower bound on the probability that an arbitrary head processor P is on-time at i:

$$\begin{aligned} &Pr[P \text{ is on-time at } i \mid Q] \\ &\geq Pr[P \text{ updated to} \geq 2 \text{ processors on-time at } i-1 \mid Q]. \end{aligned}$$

For any events A and B, we know that $Pr[A|Q] \geq Pr[A|B \wedge Q] \cdot Pr[B|Q]$. Here we let A be the event that during phase i P updated to cells of at least two processors that were on-time at $i-1$, and B be the event that P took at least $2w+1$ steps during phase i. It will now suffice to determine lower bounds on $Pr[A|B \wedge Q]$ and $Pr[B|Q]$. Let C be the event that S_P is allotted more than $(4w+2)|S_P|/c_*$ steps. As before, since $Pr[B|Q] \geq Pr[B|C \wedge Q] \cdot Pr[C|Q]$, it will suffice to consider $Pr[B|C \wedge Q]$, $Pr[C|Q]$, and $Pr[A|B \wedge Q]$.

CLAIM 6.9. $Pr[B|C \wedge Q] > 1 - 2c^*/(c_*(2w+1))$.

Proof. We wish to determine a lower bound on the probability that P takes at least $2w+1$ steps, given the fact that S_P gets more than $(4w+2)|S_P|/c_*$ steps. Suppose that S_P gets more than $(4w+2)|S_P|/c_*$ steps; let X be a random variable equal to the number of steps taken by P.

We can consider the choice of the processor to take the next step as a two-part process. First, we decide whether the processor is in S_P. If the processor is not in S_P, we next choose a particular processor; we do not care about the outcome. Otherwise, the processor is in S_P, and we next choose a processor from S_P. We are interested in the number of steps that P receives during phase i. If we know the number of steps that S_P receives, then the specific configuration has no bearing on the number of steps that P receives. Since the probability of P taking a step is at least c_*/n and at most c^*/n when considered among all n processors, the probability among a set of $|S_P|$ processors is at least $c_*/|S_P|$ and at most $c^*/|S_P|$. Thus $E[X] \geq 4w+2$, and

$$\begin{aligned} Pr[X \geq 2w+1] &\geq Pr[|X - E[X]| < 2w+1] \\ &= 1 - Pr[|X - E[X]| \geq 2w+1]. \end{aligned}$$

Since X is binomially distributed,

$$\mathrm{Var}(X) \leq \frac{(4w+2)|S_P|}{c_*}\left(\frac{c^*}{|S_P|}\right)\left(1 - \frac{c_*}{|S_P|}\right).$$

Chebyshev's inequality states that $Pr[|X - E[X]| \geq t] \leq \frac{\mathrm{Var}(X)}{t^2}$. In our case, we let $t = 2w+1$ to get

$$\begin{aligned} Pr[X \geq 2w+1] &\geq 1 - \frac{(4w+2)}{(2w+1)^2}\left(\frac{c^*}{c_*}\right)\left(1 - \frac{c^*}{|S_P|}\right) \\ &\geq 1 - \frac{2c^*}{c_*(2w+1)} \end{aligned}$$

as claimed. □

CLAIM 6.10. *With probability at least* $1 - w((w+1)^{i-1}e^{-\sqrt{n}})$,

$$Pr[A|B \wedge Q] \geq 1 - (\tfrac{2}{3})^w - w \cdot \tfrac{1}{3} \cdot (\tfrac{2}{3}).^{w-1}$$

Proof. Suppose that P has taken $2w+1$ or more steps during phase i. Then there will be at least w reads followed by writes. If we consider the state of the algorithm at the end of phase $i-1$, the w nodes read by P during i are arranged along some path of solid and dashed edges from P to the tail of the list. For any such path, the least progress P could make would be by reading its w immediate successors. Although the particular path followed and the particular w nodes read depend on the ordering of steps within phase i, we can determine a lower bound on $Pr[A|B \wedge Q]$ independent of such an ordering. For any path, P will be on-time at i if it took at least $2w+1$ steps and if at least two of the processors associated with its w immediate successors were on-time at $i-1$. We now apply the induction hypothesis to the processors of each of the w nodes that were read. For a particular one of these processors P_j and any configuration Q_j of predecessors of P_j at $i-1$, with probability at least $1 - (w+1)^{i-1}e^{-\sqrt{n}}$, the condition $C(P_j, i-1, Q_j)$ held at the end of phase $i-1$, and hence $Pr[P_j$ is on-time at $i-1 \mid Q_j] \geq 1/3$. It is then simple to calculate that, with probability at least $1 - w((w+1)^{i-1}e^{-\sqrt{n}})$, for all w processors the conditions are true. When all the conditions are true,

$$\begin{aligned} Pr[A|B \wedge Q] &\geq 1 - Pr[\text{none are on-time}] - Pr[\text{exactly one is on-time}] \\ &\geq 1 - (\tfrac{2}{3})^w - w \cdot \tfrac{1}{3} \cdot (\tfrac{2}{3})^{w-1} \end{aligned}$$

as claimed. □

To complete the proof of Lemma 6.8, let $\mathcal{A}$ be the condition that $Pr[A \mid B \wedge Q] \geq 1 - (\frac{2}{3})^w - w \cdot \frac{1}{3} \cdot (\frac{2}{3})^{w-1}$, and let $\mathcal{B}$ be the condition that $Pr[C|Q] \geq 1 - e^{-\sqrt{n}}$. If $\mathcal{A}$ and $\mathcal{B}$ are true, then

$$\begin{aligned} &Pr[P \text{ is on-time at } i \mid Q] \\ &\quad \geq Pr[A \mid B \wedge Q] \cdot Pr[C \mid Q] \cdot Pr[B \mid C \wedge Q] \\ &\quad \geq \left(1 - \left(\frac{2}{3}\right)^w - w \cdot \frac{1}{3} \cdot \left(\frac{2}{3}\right)^{w-1}\right)(1 - e^{-\sqrt{n}})\left(1 - \frac{2c^*}{c_*(2w+1)}\right). \end{aligned}$$

By the definition of w and by the fact that $1 - e^{\sqrt{n}} \geq 0.63$ for $n \geq 1$, this quantity is at least $1/3$. The probability of failure $Pr[\neg\mathcal{A} \vee \neg\mathcal{B}] \leq Pr[\neg\mathcal{A}] + Pr[\neg\mathcal{B}]$. Recall that by definition, the configuration at $i - 1$ being good with respect to Q means that $Pr[C \mid Q] \geq 1 - e^{-\sqrt{n}}$. Then, by making use of Claim 6.10 and Lemma 6.7, we obtain

$$\begin{aligned} Pr[\neg\mathcal{A}] + Pr[\neg\mathcal{B}] &\leq w((w+1)^{i-1} e^{-\sqrt{n}}) + e^{-\sqrt{n}} \\ &\leq (w+1)^i e^{-\sqrt{n}} \end{aligned} \tag{3}$$

and hence the condition $C(P, i, Q)$ is true with probability at least $1 - (w+1)^i e^{-\sqrt{n}}$, as claimed. □

Lemma 6.11 is a simple corollary of the previous lemma.

LEMMA 6.11. *Suppose that ℓ is a linearization chosen according to a distribution in $\mathcal{C}$. Then after $(8w+4)(c_*)^{-2} n \log n$ steps of ℓ, for each head processor P, the condition $Pr[P$ is promoted$] \geq 1/3$ is true with probability at least $1 - (w+1)^{\log n} e^{-\sqrt{n}}$.*

Proof. From Lemma 6.8, we know that after $\log n$ phases for any head processor P and any configuration Q of predecessors of P at $\log n$, the condition $C(P, \log n, Q)$ is true with probability at least $1 - (w+1)^{\log n} e^{-\sqrt{n}}$. We then determine that

$$\begin{aligned} Pr[P \text{ is promoted}] &= \sum_Q Pr[P \text{ is promoted} \mid Q] \cdot Pr[Q] \\ &\geq \frac{1}{3} \sum_Q Pr[Q] \geq \frac{1}{3} \end{aligned} \tag{4}$$

as claimed. □

We proceed to the second stage.

LEMMA 6.12. *Suppose that at the end of the first stage for each head processor P, $D(P)$, defined as "$Pr[P$ is promoted$] \geq 1/3$," is true with probability at least $1 - (w+1)^{\log n} e^{-\sqrt{n}}$. Then for a linearization chosen according to a distribution in $\mathcal{C}$, consider the running of the algorithm until all head processors are promoted. The expected maximum number of steps taken by any processor in this stage is at most $a_0 \log n$, for a constant a_0 and sufficiently large n.*

Proof. We note that if a processor reads the node of a single processor that is promoted, when it advances its node's pointer, it too is promoted. Each head processor can advance its node's pointer at most $n - \sqrt{n} < n$ times before reaching a tail processor. When the condition $D(P)$ holds for each processor, we can use the arguments developed in Lemma 6.2 to show that the expected maximum number of steps taken by any processor in this stage is at most $a_0 \log n$ for a constant a_0. The overall expected maximum is thus at most

$$a_0 \log n \cdot Pr[\text{All } D(P) \text{ true}] + n \cdot Pr[\text{One } D(P) \text{ false}].$$

It will suffice to show that $Pr[\text{One } D(P) \text{ false}] < 1/n$. By Lemma 6.11, the probability that $D(P)$ is false for a particular P is at most $(w+1)^{\log n} e^{-\sqrt{n}}$, or

$$Pr[\text{One } D(P) \text{ false}] \leq n \cdot (w+1)^{\log n} e^{-\sqrt{n}} \leq 1/n$$

for sufficiently large n. This completes the proof of Lemma 6.12. □

THEOREM 6.13. *The n-input problem of pointer jumping can be completed in $k \log n$ time, using n processors, with the distribution on linearizations chosen from $\mathcal{C}$, for some constant k.*

Proof. We prove the theorem by induction on the size of the pointer jumping problem, analyzing one stage at a time. The expected maximum number of steps taken by any processor in stage 1 is at most $c' \log n$, where c' is a constant depending only on w. By Lemmas 6.11 and 6.12, the expected maximum for stage 2 is at most $a_0 \log n$. In stage 3, we consider the progress of the tail processors. By the induction hypothesis, this requires a maximum of $k \log \sqrt{n} = (k/2) \log n$ steps. Finally, stage 4 is completed in another $O(n \log n)$ steps (by Fact 4.7), yielding an expected maximum of $c'' \log n$, for c'' a constant depending only on Fact 4.6. The total expected maximum is at most $(c' + a_0 + k/2 + c'') \log n$, and since c', a_0, and c'' are independent of k, we can choose $k \geq 2(c' + a_0 + c'')$, to complete the proof of the theorem. □

6.2. More complex delays. In the previous analysis, we considered together the behavior of all processors. In fact, we can assume that a constant number of processors (not including the anchor, defined in §7) are arbitrarily slow with respect to the other processors. In particular, a constant number of failures can be tolerated. Recall that a processor exits the computation as soon as it has updated its pointer to point to the tail of the list; we consider separately the steps taken by the slow and fast processors. To analyze the slow-down caused to the fast processors, we assume the worst case from the perspective of these processors: no slow processor takes a step before the fast processors finish their work. We modify our earlier analysis by showing that with high probability each processor takes $w+s$ steps per phase, where s is the number of slow processors. Even if a processor has to bypass all s slow processors in one phase, it will still have sufficiently high probability of reading two on-time processors. Finally we consider the number of steps taken by each slow processor, divided into those steps taken while the fast processors are active and those taken after all fast processors have exited the computation. Since the slow processors are no more likely to be allotted steps than the fast processors, there will be at most $O(\log n)$ steps of the first type. After the fast processors have all reached the tail of the list, a slow processor needs to read only one fast processor to update its pointer to the tail of the list. In the worst case, a slow processor will have s steps of the second type, for a total of $O(\log n)$.

7. Extensions. In the distributions considered in this paper, the speeds of the processors are within a constant factor of each other. Since in results discussed in this paper constant factors are unimportant, we have not been concerned with the difference between replacing a subset of processors by slower processors and replacing a subset of processors by faster processors. In this section we consider how the model could be extended by introducing a weighting function to incorporate other situations.

The weighting function arises from our wish to have our complexity measure reflect differences in intrinsic processor speeds. In the absence of such a function, replacing slow processors with faster ones might cause our measurement of time to increase, since faster processors might be likely to take more steps and thereby increase the expected maximum. We wish for our measure to reflect elapsed time; for example, we would like to be able to distinguish between the effect of replacing half the processors with faster ones and that of replacing half the processors with slower ones.

The weighting function, W, provides a mechanism for normalizing the numbers of steps taken by the various processors. In its fullest generality, the weight of a particular processor may depend on the number of steps it has taken, or the number of processors remaining active during the computation. In this case, rather than simply determining the sum of the steps taken by a processor, we determine the weighted sum, where each step is multiplied by the weight of that processor for that step. In a simple scenario where the weights of the processors remain the same throughout the computation, we can express W as a vector $(w_1, \ldots, w_p)$. We may wish to express the complexity of an algorithm with respect to the speed of a particular *anchor* processor. In the replacement of processors by slower or faster ones, the speed of the anchor is assumed not to have changed.

8. Conclusions and open problems. We have considered a framework for the asynchronous communication of processors through shared memory cells. We introduce a general model, parameterized by the choice of the underlying synchronization primitive and the distribution on possible processor schedules. The model and measure are shown to satisfy various criteria essential to the asynchronous setting.

The criteria include the application of the measure to the independence, synchronization and pointer jumping paradigms. For any choice of parameters, the complexity of an algorithm in the independence paradigm is equal to its synchronous PRAM complexity. For the atomic read/write cell and several classes of distributions, we can show that the complexity of the synchronization paradigm on n processors is in $\Theta(\log n)$. This result can be used to obtain straight-forward simulations of synchronous PRAM and circuit models.

It is possible to improve on the bounds obtained by the simulation of a synchronous PRAM algorithm; the pointer jumping algorithm analyzed here constitutes one such improvement. Using a synchronization primitive as weak as the atomic read/write cell and one of a general class of distributions on processor schedules, we obtain a running time of $O(\log n)$ for a pointer jumping problem of size n. The result implies fast algorithms for problems such as list ranking; the techniques used may be applicable to a wider class of problems.

Although the paradigms considered here form the basis of most synchronous parallel algorithms, it is possible that there are other paradigms that are particularly suited to the asynchronous environment. The identification and analysis of such paradigms will form part of the foundational work in the area. The algorithms given in this paper are designed to work quickly when the distribution on schedules corresponds to processors working at similar speeds. Of possible interest is the development of algorithms geared to other, more skewed distributions, or algorithms in which processors adapt themselves to fit the currently perceived distribution. In addition, it would be helpful to determine whether or not there exist algorithms such that one is faster under one distribution and the other faster under another. Finally, there is a need for algorithms that are robust in the sense that they work fairly quickly for a large class of distributions, and algorithm design techniques that apply to large classes of problems.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] R. Cole and O. Zajicek, *The APRAM: Incorporating Asynchrony into the PRAM Model*, Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 169–178, June 1989.

[3] ———, *The Expected Advantage of Asynchrony*, in Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 85–94, July 1990.

[4] S. Cook, C. Dwork, and R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[5] P. DYMOND AND W. L. RUZZO, *Parallel RAMs with owned global memory and deterministic context-free language recognition*, Proceedings of the 13th International Colloquium on Automata, Languages, and Programming, pp. 95–104, 1986.

[6] M. J. FISCHER AND N. A. LYNCH, *On Describing the Behavior and Implementations of Distributed Systems*, Theoret. Comput. Sci., 13 (1981), pp. 17–43.

[7] S. FORTUNE AND J. WYLLIE, *Parallelism in Random Access Machines*, in Proceedings of the 10th Annual ACM Symposium on the Theory of Computing, pp. 114–118, 1978.

[8] P. B. GIBBONS, *A more practical PRAM model*, in Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 158–168, June 1989.

[9] ———, *The asynchronous PRAM: A semi-synchronous model for shared memory MIMD machines*, Technical Report TR-89-062, International Computer Science Institute, Berkeley, California, 1989.

[10] L. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073–1086.

[11] M. P. HERLIHY, *Impossibility and universality results for wait-free synchronization*, Proceedings of the Seventh Annual ACM Symposium on the Principles of Distributed Computing, pp. 276–290, August 1988.

[12] M. P. HERLIHY AND J. M. WING, *Axioms for concurrent objects*, Proceedings of the 14th ACM Symposium on the Principles of Programming Languages, pp. 13–26, 1987.

[13] Z. M. KEDEM, K. V. PALEM, AND P. SPIRAKIS, *Efficient robust parallel computations*, Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing, pp. 138–148, May 1990.

[14] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *Efficient synchronization of multiprocessors with shared memory*, Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing, pp. 218–228, 1986.

[15] L. LAMPORT, *On Interprocess Communication I: Basic Formalism*, Distributed Computing, 1 (1986), pp. 77-85.

[16] ———, *On Interprocess Communication II: Algorithms*, Distributed Computing, 1 (1986), pp. 86-101.

[17] G. LEV, N. PIPPENGER, AND L. VALIANT, *A Fast Parallel Algorithm for Routing in Permutation Networks*, IEEE Transactions on Computing, C-30 (1981), pp. 93–100.

[18] C. MARTEL, A. PARK, AND R. SUBRAMONIAN, *Optimal Asynchronous Algorithms for Shared Memory Parallel Computers*, Report CSE-89-8, Division of Computer Science, University of California, Davis, CA, July 1989.

[19] N. NISAN, *CREW PRAMs and decision trees*, in Proceedings of the 21st Annual ACM Symposium on the Theory of Computing, pp. 327–335, May 1989.

[20] N. NISHIMURA, *Asynchronous shared memory parallel computation*, in Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 76–84, July 1990.

[21] ———, *Asynchrony in Shared Memory Parallel Computation*, Ph.D. thesis, University of Toronto, Toronto, Canada, 1991.

[22] P. RAGHAVAN, *Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs*, J. Comput. System Sciences, 37 (1988), pp. 130–143.

[23] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.

[24] R. E. TARJAN AND U. VISHKIN, *Finding biconnected components and computing tree functions in logarithmic parallel time*, SIAM J. Comput., 14 (1985), pp. 862–874.

[25] L. G. VALIANT, *Bulk-synchronous parallel computers*, in Parallel Processing and Artificial Intelligence, C. M. Reeve, ed., Wiley, New York, 1989, pp. 15–22.

[26] ———, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, Volume A, J. Van Leeuwen, ed., North Holland, Amsterdam, pp. 943–972, 1990.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1253–1265, December 1994

009

TIGHT BOUNDS ON OBLIVIOUS CHAINING*

SHIVA CHAUDHURI†

Abstract. The *chaining* problem is defined as follows. Given values $a_1, \ldots, a_n$, $a_i = 0$ or 1, $1 \le i \le n$, compute $b_1, \ldots, b_n$ such that $b_i = \max\{j \mid a_j = 1, j < i\}$. (Define $\max\{\} = 0$.) The chaining problem appears as a subproblem in many contexts. There are known algorithms that solve the chaining problem on CRCW PRAMs in $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse of Ackerman's function, and is a very slowly growing function. The author studies a class of algorithms (called oblivious algorithms) for this problem. A simple oblivious chaining algorithm running in $O(\alpha(n))$ time is presented. More importantly, the optimality of the algorithm is demonstrated by showing a matching lower bound for oblivious algorithms using n processors. The first steps toward a lower bound for all chaining algorithms are also provided by showing that any chaining algorithm that runs in two steps must use a superlinear number of processors. The proofs use *prefix graphs* and *weak superconcentrators*. An interesting connection between the two is demonstrated and this idea is used to obtain improved bounds on the size of prefix graphs.

Key words. parallel, chaining, superconcentrators, lower bound, prefix graphs, Ackerman's function

AMS subject classifications. 68Q20, 68Q25

1. Introduction.

Consider the following problem called *chaining*. Given values $a_1, \ldots, a_n$, $a_i = 0$ or 1, $1 \le i \le n$, compute $b_1, \ldots, b_n$ such that $b_i = \max\{j \mid a_j = 1, j < i\}$. (Define $\max\{\} = 0$.) The output can be viewed as pointers that chain the 1s into a linked list. The chaining problem is a natural problem to consider in the context of database retrieval operations; all the records that satisfy a particular predicate correspond to the input bits that have value 1. Chaining the 1s then corresponds to making a linked list of these records for future processing. Apart from this it appears as a subproblem in many contexts and has been studied before in [16] and [17]. Parallel integer sorting [2], [14], parallel merging of integers drawn from a restricted domain [3], parallel subset compaction [18], [13], [16], and circuits for computing threshold functions [15] are examples. It is easy to solve the problem in $O(n)$ time using one processor. Using n processors, very fast parallel algorithms exist, with running times close to constant. For this reason, and because of its simplicity, it is an open question of theoretical interest [3], [16], [17] whether constant time parallel algorithms exist.

Berkman and Vishkin [4] and Ragde [16] have given parallel algorithms that solve the chaining problem in $O(\alpha(n))$ time using n processors, where $\alpha(n)$ is the inverse of Ackerman's function and is a very slowly growing function. Using algorithms similar to the chaining algorithm, Berkman and Vishkin [5] give algorithms achieving the same bounds for other problems: The lowest-common-ancestor problem and a parenthesis matching problem.

We study oblivious algorithms for the chaining problem. Informally, an oblivious algorithm is one in which the pattern of memory access depends only on n (the size of the problem), and not on the specific input. This class of algorithms is of interest because the algorithms of Berkman and Vishkin and Ragde can be modified to be oblivious. We present a simple oblivious algorithm for chaining running in $O(\alpha(n))$ time. While the performance bounds are the same as previously known algorithms, our algorithm is simple and makes use of previously known graph structures. More importantly, we show that for the class of oblivious algorithms this is optimal, by proving that an oblivious chaining algorithm using n processors requires $\Omega(\alpha(n))$ time. Since all known algorithms for chaining can be made oblivious, this gives evidence of a superconstant lower bound for all chaining algorithms.

*Received by the editors April 29, 1992; accepted for publication (in revised form) August 3, 1993.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. Current address, MPI Informatik, Im Stadtwald, 66123 Saarbrücken, Germany (shiva@mpi-sb.mpg.de).

Chandra, Fortune, and Lipton [7] showed that a circuit of bounded depth for the prefix-carry problem requires superlinear size, implying a superconstant lower bound on depth for circuits with linear size. Essentially they show that such a circuit must have the structure of a special type of graph called a prefix graph. They then prove the upper and lower bounds on the size of prefix graphs of bounded depth [6], [7]. We demonstrate a connection between prefix graphs and another family of graphs called *weak superconcentrators* [11]. Using this idea, we present a simple proof that improves the lower bound of [7], and shows that the construction in [6] is optimal.

Dolev, Dwork, Pippenger, and Wigderson [11] showed a lower bound on weak superconcentrators of bounded depth. Our lower bound for oblivious chaining algorithms is obtained by interpreting such algorithms as graphs and using the techniques of [11] to analyze their properties. It is worth noting that there are chaining algorithms whose graphs are not weak superconcentrators, hence the lower bound of [11] cannot be used directly.

We provide the first steps toward proving a lower bound for all algorithms by showing that any chaining algorithm that terminates in two steps requires a superlinear number of processors. At the time of submission of this paper, we conjectured that the techniques developed in this paper would be useful in proving a lower bound for all algorithms. This was indeed the case, as the techniques used here were recently extended to prove a lower bound for all chaining algorithms in [9].

The model of computation used in this paper is the Concurrent-Read Concurrent-Write Parallel Random Access Machine (CRCW PRAM). In the COMMON model of CRCW PRAM, all processors that simultaneously write to the same memory cell must write the same value. In the more powerful PRIORITY model, each processor has an associated priority and when several processors simultaneously write to the same memory cell, the highest priority processor succeeds. It has been shown that the PRIORITY model is strictly stronger than the COMMON model [1]. We show that when considering oblivious algorithms, both models are equivalent if the COMMON machine has some extra memory. Thus, throughout this paper, the algorithms described run on the COMMON model and the lower bound is proved on the PRIORITY model.

2. Oblivious computation on PRAMs. The input to an algorithm for chaining consists of a value n (the size of the problem), and n bits (called the input vector) representing the problem. At any step each processor computes, based on its actions so far, a memory address to access and the contents to write (if it is a write step). By an *oblivious* algorithm, we mean one where the address accessed is fixed over all input vectors, i.e., it depends only on the value n. However, whether or not the processor performs any action may depend on the input vector; for example, when $n = 100$, on the fifth write step, p_1 may or may not write depending on the input vector, but if it does write, it always writes to c_{15}. (Our definition of oblivious algorithms coincides with what is called *semi-oblivious* in [10].)

It will be convenient to model the computation of an oblivious algorithm on a graph. Given an algorithm A and an input size n, the directed graph $G_{A,n}$ is defined as follows. The vertices of $G_{A,n}$ are grouped into levels. Suppose the algorithm solves the chaining problem in k steps. Then the graph $G_{A,n}$ will consist of $2k + 1$ levels of vertices, numbered $0, \dots, 2k$.

At even levels we will have a vertex for each cell in the memory that is accessed by the algorithm. These vertices will have the form $(c, 2j)$, $0 \leq j \leq k$ and will be called cell vertices (or just cells). At odd levels we will have a vertex for each processor. These vertices will have the form $(p, 2j + 1)$, $0 \leq j \leq k - 1$, and will be called processor vertices (or just processors). Thus there are k levels of processor vertices and $k + 1$ levels of cell vertices.

Edges of $G_{A,n}$ are defined as follows.

$(c, 2j) \rightarrow (p, 2j + 1)$: if for some input vector, at step $j + 1$, processor p reads cell c.

$(p, 2j+1) \rightarrow (c, 2(j+1))$: if for some input vector, at step $j+1$, processor p writes to cell c.

$(p, 2j+1) \rightarrow (p, 2(j+1)+1)$.

$(c, 2j) \rightarrow c(2(j+1))$.

Initially, bit i of the input vector is assumed to be in cell i, $1 \leq i \leq n$, and finally the output value corresponding to bit i is assumed to be in cell i. We shall refer to vertices $(i, 0)$ as x_i and vertices $(i, 2k)$ as y_i.

Let P be the number of processors used by A. The number of cells accessed during the computation is at most $2kP$. There are two kinds of edges in the graph, those adjacent to a processor vertex and others. Since each processor vertex has at most degree 4, the former are at most $4kP$ in number. For each cell accessed during the computation, there are k cell vertices in the graph, one at each even level. Thus the total number of cell vertices in the graph is $2k^2P$. Since each cell vertex is adjacent to at most two of the latter type of edge, there are at most $2k^2P$ such edges. Hence the number of edges in the graph is certainly upperbounded by $6k^2P$.

Given an input vector D of length n, we shall associate with each cell vertex a content and with each processor vertex a state. The content associated with $(c, 2j)$ is the content of cell c at step j (just before the $(j+1)$st read-write step) in the computation of A on input D. The state associated with $(p, 2j+1)$ is the state of the processor p after the read step of the $(j+1)$th read-write step in the same computation. At any time in the computation, the action of a processor is dependent solely on its state. Each processor starts with a fixed initial state.

A *partial input* $B = b_1b_2 \ldots b_n$ is one in which each $b_i \in \{0, 1, *\}$. An input vector $D = d_1d_2 \ldots d_n$, each $d_i = 0$ or 1, is consistent with B if $\forall i$, $1 \leq i \leq n$, $b_i \neq * \Longrightarrow b_i = d_i$. X_B will denote the set of inputs consistent with B. For a partial input B and a cell vertex $(c, 2j)$ define

$$S_B((c, 2j)) = \{d \mid \text{ cell } (c, 2j) \text{ has content } d \text{ for some } x \in X_B\}.$$

Similarly, for a processor vertex $(p, 2j+1)$, define

$$S_B((p, 2j+1)) = \{e \mid (p, 2j+1) \text{ has state } e \text{ for some } x \in X_B\}.$$

We say a (cell or processor) vertex (x, j) is *fixed* for a partial input B if $|S_B((x, j))| = 1$.

2.1. Oblivious COMMON simulation of oblivious PRIORITY. Consider the following problem called the *leftmost-one* problem. Given input $B = b_1, \ldots, b_n, b \in \{0, 1\}$, compute $s = \min\{j \mid b_j = 1\}$. Fich, Ragde, and Wigderson [12] show that the leftmost-one problem can be solved in $O(1)$ time on COMMON using n processors and n memory cells. Their algorithm can be made oblivious. We shall use this fact to prove the following.

LEMMA 2.1. *Consider an oblivious algorithm that runs on a* PRIORITY PRAM *with p processors and M memory cells, in k steps. Then there is an oblivious algorithm solving the same problem on a* COMMON PRAM *with p processors and M + p memory cells in $O(k)$ steps.*

Proof. It suffices to show that one write step of an oblivious PRIORITY algorithm can be simulated on an oblivious COMMON machine in $O(1)$ steps. Suppose the PRIORITY machine writes to r cells $c_1, \ldots, c_r$. Let D_i denote the set of processors that may write to c_i. D_i is a fixed set. Note that some of the processors may choose not to write. It is sufficient for the COMMON machine to find, for each i, the highest priority processor in D_i that chooses to write. This is done by solving a leftmost-one problem of size $|D_i|$, using $|D_i|$ cells, processors from D_i, and $O(1)$ time. The space bound follows from the fact that $\sum_{i=1}^{r} |D_i| \leq p$. □

Henceforth we shall refer only to PRIORITY algorithms, and by the lemma, all the algorithms run on COMMON with the same time bounds. Note that in general it is not true that one step of a PRIORITY algorithm can be simulated by a COMMON algorithm in $O(1)$ steps. Boppana [1] gives an example of a problem that can be solved in $O(1)$ time on PRIORITY but requires $\Omega(\frac{\log n}{\log\log n})$ time on COMMON.

3. Upper bounds. Ackerman's function is defined as follows:

$$A(0,0) = 0;\; A(i,0) = 1 \quad \text{for } i > 0;\; A(0,j) = 2j \quad \text{for } j > 0,$$
$$A(i,j) = A(i-1, A(i,j-1)).$$

For a function f let $f^{(1)}(n) = f(n)$; $f^{(i)}(n) = f(f^{(i-1)}(n))$, $i > 1$. Define $I_0(n) = \lceil \frac{n}{2} \rceil$ and $I_k(n) = \min\{j \mid I_{k-1}^{(j)}(n) \leq 1\}$, $k > 1$. The functions I_k are the inverses of the kth level of Ackerman's function, i.e., $I_k(n) = \min\{j \mid A(k,j) \geq n\}$. I_1 behaves like $\log n$ and I_2 like $\log^* n$. Define $\alpha(n) = \min\{j \mid I_j(n) \leq j\}$.

Berkman and Vishkin et al. [4], [3], and Ragde [16] have given algorithms that solve the chaining problem on PRIORITY in ck steps using $nI_k(n)$ processors, where c is a constant ≥ 2. From these algorithms one can construct an algorithm using $O(n)$ processors that takes $O(\alpha(n))$ time. We give simple oblivious algorithms that solve the problem in $2k$ steps on PRIORITY using $nI_k(n)$ processors. Though the performance bounds are the same, we feel our algorithm is easier to understand.

A *prefix graph* of size n is a directed acyclic graph with n vertices $(x_1, \ldots, x_n)$ of indegree 0 called input vertices and n vertices $(y_1, \ldots, y_n)$ of outdegree 0 called output vertices. The depth of a prefix graph is the length of the longest path from an input to an output. Prefix graphs have the following property: $\forall i, j \in [n]$, there is a directed path from x_i to y_j iff $i < j$. Say a prefix graph is *levelled* if the vertices can be partitioned into levels numbered $0, 1, 2, \ldots$, such that every edge is from a level $i-1$ vertex to a level i vertex for some $i \geq 1$. Call such an edge a level i edge. A prefix graph is *contiguous* if for any vertex v the inputs from which v is reachable are of the form x_r, $r \in [i,j]$. It is possible to construct [6] levelled contiguous prefix graphs of size n and depth $2k$ such that $\forall i$, $1 \leq i \leq 2k$, the number of level i edges $\leq nI_k(n)$.

The *restricted-domain prefix-maxima* problem is defined as follows: Given an input $a_1, \ldots, a_n$; $a_i \geq 0$, $1 \leq i \leq n$, and $\forall i, j$, $1 \leq i < j \leq n, a_i, a_j \neq 0 \Longleftarrow a_i < a_j$, compute $b_1, \ldots, b_n$ where $b_i = \max\{a_j \mid j < i\}$. We show how to solve this problem using a prefix graph. Initially set the value at vertex $x_i = a_i$, $1 \leq i \leq n$. At step i, level i edges propagate the values at their tails to their heads and vertices at level i select the largest value propagated to them. It is easy to see that the value at a vertex $v = \max\{a_j | v \text{ is reachable from } x_j\}$ and thus the value at $y_i = b_i$, $1 \leq i \leq n$.

THEOREM 3.1. $\forall k \geq 1$, *there is an oblivious* PRIORITY PRAM *algorithm using* $nI_k(n)$ *processors that solves the chaining problem in* $2k$ *steps.*

Proof. We show how an oblivious PRIORITY PRAM algorithm can simulate the computation of a levelled, contiguous prefix graph G. Label the edges of G as follows. Let $\{x_r | r \in [i,j]\}$ be the set of input vertices that can reach the vertex at the tail of edge e. Label e with j. Designate a memory cell $m(v)$ to correspond to each vertex v of G. At step i, allocate a processor $p(e)$ to each level i edge of G so that for any two edges e, f such that label$(e) <$ label(f), $p(f)$ has a higher priority than $p(e)$. Such an allocation is easy to do. For e, an edge from v to w, $p(e)$ reads the value in $m(v)$. If the value is 0, then $p(e)$ does nothing, otherwise it writes the value to $m(w)$. It is easy to show that the value in $m(w)$ is $\max\{a_r | w \text{ is reachable from } x_r\}$, and so $m(y_i) = b_i$, $1 \leq i \leq n$. To solve the chaining problem with input $a_1, \ldots, a_n$, at first, $1 \leq i \leq n$, p_i writes i into $m(x_i)$ if $a_i = 1$. Then simply solve the restricted-domain prefix-maxima problem with the values in $m(x_i)$, $1 \leq i \leq n$ as input. The stated bounds follow from the bounds on prefix graphs. □

3.1. Weak superconcentrators and prefix graphs. In [7], it is proved that a prefix graph of depth $2k$ requires $\Omega(nI_{2k-1}(n))$ edges. We improve the lower bound, showing that the construction is optimal.

A *weak superconcentrator* is a directed acyclic graph with n vertices $x_1, \ldots, x_n$ of indegree 0 and n vertices $y_1, \ldots, y_n$ of outdegree 0, and the property that $\forall k$ and $i_1 < j_1 < i_2 < j_2 < \cdots < i_k < j_k$, there exist vertex disjoint paths between $\{x_{i_1}, \ldots, x_{i_k}\}$ and $\{y_{j_1}, \ldots, y_{j_k}\}$. The depth of a weak superconcentrator is the length of the longest directed path in it. It is proved in [11] that a weak superconcentrator of depth $2k$ requires $\Omega(nI_k(n))$ edges.

THEOREM 3.2. *A prefix graph of depth* $2k$ *requires* $\Omega(nI_k(n))$ *edges.*

Proof. We show that every prefix graph is a weak superconcentrator. Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ be the input and output vertices of a prefix graph G. $\forall k$ and $i_1 < j_1 < i_2 < j_2 < \cdots < i_k < j_k$ there exist paths from x_{i_m} to y_{i_m}. If they are not all vertex disjoint, then $\exists p, q,\ 1 \leq p < q \leq k$ such that the paths from x_{i_p} to y_{j_p} and x_{i_q} to y_{j_q} have a common vertex. But then there is a path from x_{i_q} to y_{j_p}, $i_q > j_p$, a contradiction. □

4. Some useful functions and their properties. Following [11], we define the trees $T_k(l),\ l \geq 1$. $T_k(l)$ has all its leaves at depth k and each edge is labelled with a power of 2. The outdegree of the root is l and the outdegree of every other vertex is the label of the edge coming into the vertex from its parent. We describe how to construct $T_k(l)$, starting from the root. The edges and vertices of $T_k(l)$ will be created in a certain order. We think of the vertices at a given depth as being arranged from left to right in order of creation. The rule governing the labelling of edges is, at any depth, the label of the first edge created is 1 and the label of the jth edge created is twice the maximum of the product of labels on a path starting with the $(j-1)$th edge created and ending at a leaf.

Initially we are at the root. When we are at a vertex v at depth less than k, and the number of children of v created so far is less than the label of the edge to v from its parent (or less than l, if v is the root), then we create a new child of v, label the connecting edge as per the rule, and move to the new child. If the number of children created so far is equal to the label on the edge to v from its parent, we move to the parent of v. If v is the root and it already has l children, the construction is complete. When we are at a vertex at depth k, i.e., a leaf, we simply move back to the parent.

Recall the definition of Ackerman's function from the previous section. In the Appendix, it is shown that the maximum of the product of the labels on a path in $T_k(l)$, from the root to a leaf is at most $A(k, 2l)$. Set $l = \lfloor \frac{I_k(n)-1}{2} \rfloor \geq \frac{I_k(n)}{3}$ for n sufficiently large. Let H be the set of leaves. For $h \in H$, let $c_1(h), \ldots, c_k(h)$ be the labels on the edges of the path from the root to h, in that order. Then the following inequalities hold. (1a)–(1f) are similar to inequalities that were proved before in [11] and [8]. We include their proofs in the Appendix. We shall prove the others here.

$$\forall h \in H,\ c_1(h) \geq 1, \ldots, c_k(h) \geq 1, \tag{1a}$$

$$\forall h \in H, c_1(h) \ldots c_k(h) \leq n, \tag{1b}$$

$$\sum_{h \in H} c_k(h) \leq 2n, \tag{1c}$$

$$\sum_{h \in H} \frac{1}{c_1(h) \ldots c_k(h)} \leq 2, \tag{1d}$$

$$1 \le j \le k-1, \quad \sum_{h \in H, c_1(h) \dots c_k(h) < y} c_1(h) \dots c_{j-1}(h) c_k(h) \le 2y, \tag{1e}$$

$$\sum_{h \in H} \frac{1}{c_1(h) \dots c_{k-1}(h)} \ge \frac{I_k(n)}{3}, \tag{1f}$$

$$\exists y_0(k), \text{ s.t. } \forall y > y_0(k), \ 1 \le j \le k-1, \tag{1g}$$

$$\sum_{h \in H, \ c_j(h) < y \le c_1(h) \dots c_k(h)} \frac{1}{c_j(h) \dots c_{k-1}(h)} \le 2,$$

$$\forall y > 0, \ 1 \le j \le k-1, \quad \sum_{h \in H, \ c_j(h) < y \le c_1(h) \dots c_k(h)} \frac{1}{c_j(h) \dots c_{k-1}(h)} \le y. \tag{1h}$$

Fact. Let v be a nonleaf node in the tree $(T_k(l))$ and let w be the next node to its right at the same level. Let c and d be the labels of the edges to v and w from their parent(s), respectively, and let e be the label of the edge from v to the rightmost child of v. Then it is easily seen that $e \ge 2^c$ and $d > e \ge 2^c$.

LEMMA 4.1. *Let $c_1, \dots, c_k$ be the labels on some path from the root to a leaf. Then* $1 \le i \le k-1, \ c_i \le c_{i+1}^2$.

Proof. Let $h_1, \dots, h_{|H|}$ be the leaves of the tree, from left to right. The lemma clearly holds for the path to h_1 (all the labels are 1). Assuming the lemma holds for the path to h_m, we shall show that it holds for the path to h_{m+1}.

The paths to h_m and h_{m+1} diverge at some level; call this level r. Then $c_1, \dots, c_r$ are common to both levels. Let $c_{r+1}, \dots, c_k$ and $d_{r+1}, \dots, d_k$ be the remaining labels on the paths to h_m and h_{m+1}, respectively. Note that each of $c_{r+2}, \dots, c_k$ are labels to rightmost children, so by the fact mentioned previously, $c_{j+1} \ge 2^{c_j}$ for $r+1 \le j \le k-1$. By the inductive hypothesis, $\forall j, 1 \le j \le r-1, \ c_j \le c_{j+1}^2$. Since $d_{r+1} > c_{r+1}$ and $c_r \le c_{r+1}^2$, we have $c_r \le d_{r+1}^2$. Now $\forall j, \ r+1 \le j \le k-1, d_j = 2c_j \dots c_k \le 22^{c_j} c_{j+1} \dots c_k \le 2c_{j+1}^2 c_{j+2} \dots c_k \le 4(c_{j+1} \dots c_k)^2 = d_{j+1}^2$. This completes the proof. □

LEMMA 4.2. *There is a function $y_0(k)$ such that the following holds. Let l, y, j be positive integers. Let $1 \le j \le k-1$. Let $y > y_0(k)$. Consider the set of paths from the root to a leaf, which have the property that $c_j < y \le c_1 \dots c_k$. Let $S_j, \dots, S_k$ be the set of vertices they pass through at levels $j, \dots, k$, respectively. Then $|S_j| \le 2$.*

Proof. Let $v_1, \dots, v_{|s_j|}$ be the vertices in S_j, from left to right. Consider the path to the leftmost leaf, among the set of considered paths. This path passes through v_1. Let the labels on this path be $c_1, \dots, c_k$.

If $c_j \ge \log \ y$, then the label of the edge to the vertex to the right of v_1 is at least y, and we are done. So assume $c_j < \log \ y$. Then by Lemma 4.1, $1 \le i \le j-1, \ c_i \le (\log \ y)^{2^{j-i}}$, i.e., $c_1 \dots c_{j-1} \le (\log \ y)^{2^j} \le (\log \ y)^{2^k}$. Since $c_1 \dots c_k \ge y$, we have $c_j \dots c_k \ge y/((\log y)^{2^k})$. Let $2 \le i \le |S_j|$, d_i be the label of the edge to the $i-1$th vertex to the right of v_1. We have $d_2 = 2c_j \dots c_k \ge 2y/((\log y)^{2^k})$ and from the fact mentioned previously, $d_3 \ge 2^{2y/((\log y)^{2^k})}$. Define $y_0(k) = \min\{x \mid 2^{2x/((\log x)^{2^k})} > x\}$. For $y > y_0(k), d_3 > y$, hence $|S_j| \le 2$. □

COROLLARY 4.3 (Inequalities (1g), (1h)). *If $y > y_0(k)$,*

$$\sum_{h \in H, \ c_j(h) < y \le c_1(h) \dots c_k(h)} \frac{1}{c_j(h) \dots c_{k-1}(h)} \le 2.$$

If $y \le y_0(k)$,

$$\sum_{h\in H,\ c_j(h)<y\le c_1(h)\dots c_k(h)} \frac{1}{c_j(h)\dots c_{k-1}(h)} \le y.$$

Proof. For v a vertex at level j of $T_k(l)$, let $c_1(v), \dots, c_j(v)$ be the labels on the path from the root to v, in that order. Then we have

$$\sum_{h\in H,\ c_j(h)<y\le c_1(h)\dots c_k(h)} \frac{1}{c_j(h)\dots c_{k-1}(h)} = \sum_{s\in S_k} \frac{1}{c_j(h)\dots c_{k-1}(h)}$$

$$\le \sum_{s\in S_{k-1}} \frac{1}{c_j(h)\dots c_{k-2}(h)} \le \cdots \le \sum_{s\in S_j} 1 = |S_j|.$$

From Lemma 4.2 we have $|S_j| \le 2$ if $y > y_0(k)$. On the other hand, since the labels on successive edges at the same level increase by at least a factor of two, there can be at most $\log\ y \le y$ edges at level j before the label exceeds y. □

5. The lower bound. Consider the following facts. Theorem 3.1 shows how to obtain an oblivious chaining algorithm whose graph (as defined in §2) is a prefix graph. By Theorem 3.2, every prefix graph is a weak superconcentrator. A lower bound for weak superconcentrators is known from [11]. Given these facts, it is tempting to conjecture that the graph of every oblivious chaining algorithm is a prefix graph and thereby directly obtain a lower bound. However, there are chaining algorithms whose graphs are not weak superconcentrators. As an example, consider an algorithm in which, in the first two steps, processor P_1 reads $x_1$and x_2 without writing anywhere. In the next two steps, P_1 writes the values of x_1 and x_2 to cells z_1 and z_2, respectively. Now the chaining problem is solved for input $z_1, z_2, x_3, \dots, x_n$, without any processor ever reading cells x_1 and x_2 again. In the graph of this computation, all paths from x_1 and x_2 must pass through vertex $(P_1, 3)$ and hence are never vertex disjoint. Thus the graph is not a weak superconcentrator. This simple example generalizes. It is necessary, therefore, to carefully analyze the structure of the graph of a chaining algorithm.

We now prove that a PRIORITY algorithm that solves the chaining problem with n processors requires $\Omega(\alpha(n))$ time. It suffices to prove the following.

THEOREM 5.1. *For n sufficiently large, any oblivious* PRIORITY *algorithm that solves a chaining problem of size n in k steps requires $\Omega(nI_{k+1}(n))$ processors.*

Proof: Fix n. Let $G_{A,n}$ be the graph for algorithm A, which terminates in k read-write steps, and let $x_1, \dots, x_n$ and $y_1, \dots, y_n$ be the input and the output vertices of $G_{A,n}$. Let H be the set of leaves of $T_k(l)$ defined in the previous section, with $l = \lfloor \frac{I_k(n)-1}{2} \rfloor \ge \frac{I_k(n)}{3}$. Fix $h \in H$ and let $p(h) = \frac{1}{c_1(h)\dots c_k(h)}$. Pick U, a random subset of $[n]$, by picking each element of $[n]$ independently with probability $p(h)$. Consider the partial input $B = b_1 \dots b_n$, $b_i = 0$ if $i \in [n] - U$, and $b_i = *$ if $i \in U$. Fix the values of the input vertices of $G_{A,n}$ as indicated by B. For a vertex v of $G_{A,n}$ let f_v denote the indegree of v. Let V_j denote the set of vertices at level $2j$, $0 \le j \le k$. Call a vertex $v \in V_j$, $1 \le j \le k-1$, *high degree* if $f_v > c_j(h)$.

Consider a high-degree vertex $(c, 2j)$ which is not fixed for B, and let $(p, 2j-1)$ be the highest priority processor that writes to $(c, 2j)$ over all inputs consistent with B, i.e., there is a setting for the variables $\{b_i \mid i \in U\}$ so that p writes to c at step j. The state of p can be affected only by those input vertices that can reach $(p, 2j-1)$. Modify B and the input vertex settings so that over all inputs consistent with the new B, $(p, 2j-1)$ writes some fixed value to $(c, 2j)$. Now $(c, 2j)$ is fixed for B, since $(p, 2j-1)$ will override any other processor that writes. If no processor writes to $(c, 2j)$, then modify B by setting to 0 all the variables that reach $(c, 2(j-1))$. Then $(c, 2(j-1))$ is fixed, and so is $(c, 2j)$. Call this operation *fixing* a vertex.

Carry out the following two-step procedure on $G_{A,n}$.

(A) For j starting at 1 and going up to $k-1$, fix all the high-degree vertices in V_j.

(B) For i starting at 1 and going up to n, if $i \in U$, set all the input vertices (with value $*$) that can reach y_i through low-degree vertices to the value 1, except x_i.

We claim that at the end of this procedure all the undefined input vertices are fixed, except possibly one. To see this, suppose more than one input vertex is undefined. Set all to 1 except the leftmost undefined input vertex x_i. Let this partial input be C. Let x_j be the first input vertex to the right of x_i that has the value 1. Clearly, x_j exists and $i, j \in U$. Since x_i was not set in step (B), x_i does not reach y_j through low-degree vertices. However, for the two inputs consistent with C, only vertices reachable from x_i through low-degree vertices can have different values, since the high-degree vertices are fixed. Hence the state of y_j remains fixed over both inputs, an error.

Write $E_{A,B}$ for the expected number of input bits that are set during this process. By the above argument, $E_{A,B} \geq E(|U|) - 1 \geq np(h) - 1$. We now obtain an upper bound for $E_{A,B}$.

Note that when fixing $v \in V_j$, all the high-degree vertices in V_i, $1 \leq i < j$ have already been fixed. Thus, the only variables that affect the contents of v are the undefined variables that can reach v through low-degree vertices. Hence $D_v := E(\text{number of inputs set to fix } v \in V_j) \leq E(\text{number of undefined inputs that can reach } v \text{ through low-degree vertices}) \leq f_v 2c_{j-1}(h)2c_{j-2}(h)\dots 2c_1(h)p(h)$. When we actually fix $v \in V_j$, we either fix the state of the highest priority processor, p, that writes to v, or if no processor writes, we fix v', the vertex that represents the same memory cell as v at the previous time step. The number of inputs fixed is at most the number of inputs that reach either of the two through low-degree vertices, i.e., at most $2c_{j-1}(h)2c_{j-2}(h)\dots 2c_1(h) = 2^{j-1}c_1(h)\dots c_{j-1}(h)$.

For a high-degree vertex $v \in V_j$, D_v is at most the minimum of these two quantities, i.e., $D_v \leq 2^{j-1}c_1(h)\dots c_{j-1}(h)\min[f_v p(h), 1]$. The expected number of bits set during (A) is the sum of the expected number of bits set while fixing each high-degree vertex, i.e., $E_A := E(\text{number of bits set during (A)}) \leq \sum_{j=1}^{k-1}\sum_{v\in X(j)} D_v$, where $X(j) = \{v \in V_j \mid c_j(h) < f_v\}$. Thus,

$$E_A \leq \sum_{j=1}^{k-1}\sum_{X(j)} D_v \leq \sum_{j=1}^{k-1}\sum_{X(j)} 2^{j-1}c_1(h)\dots c_{j-1}(h)\min[f_v p(h), 1].$$

We now upper bound E_B, the expected number of inputs set during step (B). Let y_j be an output vertex and let $S = \{x_i \mid x_i \text{ can reach } y_j \text{ through low-degree vertices and } i \in U,\ i \neq j\}$. As before, $|S| \leq f_{y_j}2c_{k-1}(h)2\dots c_1(h)$. For $x_i \in S$, P(x_i is set while processing y_j in step (B))$\leq$ P($i \in U$ and $j \in U$). Since $i \neq j$, the events $i \in U$ and $j \in U$ are independent and P($i \in U$ and $j \in U$) $= p(h)^2$. Hence E(number of inputs set while processing y_j in step (B)) $\leq f_{y_j}2c_{k-1}(h)2\dots c_1(h)p(h)^2$. As before, E_B is the sum, over all output vertices y_i, $i \in U$, of these expectations. So $E_B \leq \sum_{v\in V_k} 2^{k-1}f_v c_1(h)\dots c_{k-1}(h)p(h)^2$. Clearly, $E_{A,B} = E_A + E_B$. Thus,

$$\sum_{j=1}^{k-1}\sum_{X(j)} 2^{j-1}\min\left[\frac{f_v}{c_j(h)\dots c_k(h)}, c_1(h)\dots c_{j-1}(h)\right]$$

$$+ \sum_{v\in V_k} 2^{k-1}\frac{f_v}{c_k(h)}p(h) \geq E_A + E_B = E_{A,B} \geq np(h) - 1.$$

Multiplying both sides by $c_k(h)$, summing over $h \in H$, and interchanging the order of summation, we get

$$\sum_{j=1}^{k-1}\sum_{v\in V_j}\sum_{Y(v,j)} 2^{j-1}\min\left[\frac{f_v}{c_j(h)\dots c_{k-1}(h)}, c_1(h)\dots c_{j-1}(h)c_k(h)\right] + \sum_{v\in V_k}\sum_{h\in H} 2^{k-1} f_v p(h)$$
$$\geq \sum_{h\in H}\left[\frac{n}{c_1(h)\dots c_{k-1}(h)} - c_k(h)\right],$$

where $Y(v, j) = \{h \in H \mid c_j(h) < f_v\}$.

Let $Z(v, j) = \{h \in H \mid c_j(h) < f_v \leq c_1(h)\dots c_k(h)\}$ and $W(v, j) = \{h \in H \mid c_1(h)\dots c_k(h) < f_v\}$. Notice that $Y(v, j)$ is the disjoint union of $Z(v, j)$ and $W(v, j)$. Using this observation and rewriting, we get

$$\sum_{j=1}^{k-1}\sum_{v\in V_j} 2^{j-1}\sum_{Z(v,j)}\frac{f_v}{c_j(h)\dots c_{k-1}(h)} + \sum_{j=1}^{k-1}\sum_{v\in V_j} 2^{j-1}\sum_{W(v,j)} c_1(h)\dots c_{j-1}(h)c_k(h)$$
$$+\sum_{v\in V_k} 2^{k-1}\sum_{h\in H}\frac{f_v}{c_1(h)\dots c_k(h)} \geq \sum_{h\in H}\left[\frac{n}{c_1(h)\dots c_{k-1}(h)} - c_k(h)\right]. \tag{2}$$

Before simplifying this further, we observe that the sum of the indegrees of all the cell vertices is bounded by the number of edges and that the number of cell vertices is bounded by $2k^2P$, i.e.,

$$\sum_{j=1}^{k-1}\sum_{v\in V_j} f_v \leq 6k^2P, \tag{3a}$$

$$\sum_{j=1}^{k-1}\sum_{v\in V_j} 1 \leq 2k^2P. \tag{3b}$$

We now separately bound each of the terms of the left-hand side (LHS) of (2), which we call (I), (II), and (III), respectively. (I) can be written as the sum of two terms by separating vertices of degree greater than $I_{k+1}(n)$ and others. Thus,

$$\text{(I)} = \sum_{j=1}^{k-1}\sum_{v\in V_j,\ f_v\leq I_{k+1}(n)} 2^{j-1}\sum_{Z(v,j)}\frac{f_v}{c_j(h)\dots c_{k-1}(h)}$$
$$+\sum_{j=1}^{k-1}\sum_{v\in V_j,\ f_v> I_{k+1}(n)} 2^{j-1}\sum_{Z(v,j)}\frac{f_v}{c_j(h)\dots c_{k-1}(h)}.$$

Using (1h) and (3b) to bound the first term and (1g) and (3b) to bound the second, we find

$$\text{(I)} \leq 2^{k-1}(I_{k+1}(n))^2\cdot 2k^2P + 2^{k-1}\cdot 2\cdot 6k^2P.$$

Using (1e) and (3a) to simplify (II) we get

$$\text{(II)} \leq 2^{k-1}\cdot 2\cdot 6k^2P.$$

Term (III) can be estimated using (1d) and (3a), yielding

$$\text{(III)} \le 2^{k-1} \cdot 2 \cdot 6k^2 P.$$

The right-hand side (RHS) of equation (2) can be estimated using (1f) and (1c) to give

$$\text{RHS} \ge n\frac{I_k(n)}{3} - 2n.$$

Thus, rewriting (2) and simplifying yields

$$P = \Omega\left(\frac{nI_k(n)}{2^k k^2 (I_{k+1}(n))^2}\right) = \Omega(nI_{k+1}(n)). \qquad \square$$

6. Nonoblivious lower bounds. The techniques used in the lower bounds for oblivious algorithms may extend to nonoblivious algorithms. We conjecture similar lower bounds for all algorithms but are able to prove it only for the case $k = 2$. ($k = 1$ is easy.) Although the following theorem may be proved using computations similar to those used in the oblivious case, we use simpler computations that we hope are more illuminating. We associate a graph $G_{A,B}$ with a nonoblivious algorithm A and partial input B. As before, $G_{A,B}$ will have levels of vertices $(c, 2j)$ and $(p, 2j+1)$, $0 \le j \le k-1$. The edges are defined by the following.

$(c, 2j) \to (p, 2j+1)$: if for some input vector consistent with B, at step $j+1$, processor p reads cell c.

$(p, 2j+1) \to (c, 2(j+1))$: if for some input vector consistent with B, at step $j+1$, processor p writes to cell c.

$(p, 2j+1) \to (p, 2(j+1)+1)$.

Note that in this case we do not have edges from cell vertices to cell vertices.

For v a vertex of $G_{A,B}$ define $S_B(v)$ as before.

THEOREM 6.1. *Any* PRIORITY PRAM *algorithm that solves the chaining problem in* 2 *steps requires* $\frac{1}{8}n(\log n)^{1/2}$ *processors.*

Proof. Let $P \le \frac{1}{8}n(\log n)^{1/2}$ be the number of processors used by A. Suppose A terminates in two steps. $G_{A,B}$ consists of five levels (numbered 0,...,4) of vertices. Let $e_{B,i}$, $1 \le i \le 4$ be the set of edges between level $i-1$ and level i in $G_{A,B}$. It is easily seen that for any partial input B, $|e_{B,1}| \le P$ and $|e_{B,2}| \le 2P$.

Let $1 \le i \le 2P$, $D_i = \{v \mid v$ is a level 2 vertex with indegree $\ge i\}$, and $d_i = |D_i|$. Then we have $\sum_{i=1}^{2P} d_i \le 2P \le \frac{1}{4}n(\log n)^{1/2}$. Let $z = 2^{(\log n)/2}$. If $\forall i$, $1 \le i \le z$, $d_i > n/(2i(\log n)^{1/2})$ then $\sum_{i=1}^{z} d_i > \frac{1}{4}n(\log n)^{1/2}$, so $\exists i_0$, $1 \le i_0 \le z$ such that $d_{i_0} \le n/(2i_0(\log n)^{1/2})$.

Let $v \in D_{i_0}$ and consider the highest priority processor p that has an edge to v. The state of this processor depends only on one input bit. We set this input bit so that p writes to v. Let B be the partial input so defined. Clearly $|S_B(v)| = 1$, so v is fixed. In this manner, fix all the vertices in D_{i_0}, and let C be the partial input obtained by this procedure. The number of bits set in C is at most d_{i_0}, since, fixing each vertex in D_{i_0} involves setting at most one bit.

All the level 2 vertices of $G_{A,C}$ that are not fixed are written to by less than i_0 processors, each in at most two states. The different things that can be written to the vertex are the (at most) two values that each processor may write, and, in case no processor writes, the (at most) two values that it initially contained. Hence $\forall v$, $|S_C((v, 2))| \le 2(i_0 - 1) + 2 = 2i_0$. Since $\forall p$, $|S_C((p, 1))| \le 2$, and each processor reads a vertex that can have at most $2i_0$ different values written, the number of states of a processor is bounded by the product of the two, i.e., $\forall p$, $|S_C((p, 3))| \le 4i_0$.

Let $y_1, \ldots, y_n$ be the output vertices of $G_{A,C}$. Suppose $r = d_{i_0} \le n/(2i_0(\log n))^{1/2}$ bits $(c_{l_1}, \ldots, c_{l_r})$ have been set in C. Define $l_0 = 0$ and $l_{r+1} = n+1$. Consider m, j such that $l_j < m < l_{j+1}$. Since all the bits between c_{l_j} and c_m are undefined, y_m may have, as its final value, any one of $l_j, l_j + 1, \ldots, m - 1$, or 0, so $|S_C((y_m, 4))| \geq m - l_j + 1$. Thus $\sum_{l_j<m<l_{j+1}} |S_C((y_m, 4))| \geq \frac{1}{2}(l_{j+1} - l_j + 1)(l_{j+1} - l_j) - 1 \geq \frac{1}{2}(l_{j+1} - l_j)^2$. Since $\sum_{j=1}^{r}(l_{j+1} - l_j) \geq n$, a simple minimization shows that

$$\sum_{j=0}^{r} \sum_{l_j<m<l_{j+1}} |S_C((y_m, 4))| \geq \frac{n^2}{2(r+1)}. \tag{4}$$

For $e \in e_{C,4}$, an edge from $(p, 3)$ to $(c, 4)$, let $g(e)$ be the number of different values that p writes to c over all inputs consistent with C. For any output vertex $(y, 4)$, let e_y be the set of edges into $(y, 4)$. The number of different values that can be written to $(y, 4)$ is bounded by the number of values that different processors can write via edges in e_y plus the number of values it had before (in the case that no processor writes). Hence $S_C((y, 4)) \leq \sum_{e\in e_y} g(e) + S_C((y, 2)) \leq \sum_{e\in e_y} g(e) + 2i_0$, which yields

$$S_C((y, 4)) - 2i_0 \leq \sum_{e\in e_y} g(e). \tag{5}$$

For any processor $(p, 3)$, the total number of different values it can write over all edges leading out of it is bounded by $S_c((p, 3)) \leq 4i_0$. Summing this quantity over all processors gives an upper bound on the number of different values that can be written via edges in $e_{C,4}$. Thus

$$4i_0 P \geq \sum_{e\in e_{C,4}} g(e) \geq \sum_{i=1}^{n} \sum_{e\in e_{y_i}} g(e).$$

From equation (5), we get

$$4i_0 P \geq \sum_{i=1}^{n}(|S_C((y_i, 4))| - 2i_0) \geq \sum_{j=0}^{r} \sum_{l_j<m<l_{j+1}} |S_C((y_m, 4))| - \sum_{i=1}^{n} 2i_0.$$

Finally, from equation (4), we get

$$4i_0 P \geq \frac{n^2}{2(r+1)} - 2i_0 n.$$

Since $r \leq n/(2i_0(\log n)^{1/2})$, this yields

$$P \geq \frac{1}{4}n(\log n)^{\frac{1}{2}} - \frac{n}{2} > \frac{1}{8}n(\log n)^{\frac{1}{2}}, \text{ a contradiction.} \qquad \square$$

7. Conclusion and open problems.

We have shown that oblivious chaining with n processors is $\Theta(\alpha(n))$ time. This leaves open the question of whether an $O(1)$ time nonoblivious algorithm exists. Recently, Chaudhuri and Radhakrishnan [9] settled this question, showing an $\Omega(\alpha(n))$ lower bound for any chaining algorithm using $O(n)$ processors.

Using randomization, better performance may be achieved in some situations. Raman [17] gave a randomized algorithm that runs in $O(1)$ time if the number of 1s in the input is not too large. We conjecture that for arbitrary inputs, constant-time chaining is not possible, even using randomization.

Appendix.

LEMMA A.1 *The maximum product of labels along a path from the root to a leaf of $T_k(l)$ is at most $A(k, 2l)$.*

Proof. Let $B(k, l)$ be the maximum product of labels on a path from the root to a leaf in $T_k(l)$. Define the following auxiliary family of trees $S_k(l)$. The definition of $S_k(l)$ is the the same as $T_k(l)$ except that the first edge created at each level has the label l (instead of 1). Let $C(k, l)$ be the maximum product of labels on a path from the root to a leaf of $S_k(l)$. Then

$$B(k, l) \leq C(k, l) \leq B(k, 2l) \tag{6}$$

since we can find a subtree of $S_k(l)$ isomorphic to $T_k(l)$, such that the labels on each edge are greater than or equal to the labels on the corresponding edges of $T_k(l)$. The second inequality holds for the same reasoning with $T_k(2l)$ and $S_k(l)$.

We will now show by induction on k and l that $B(k, l) \leq A(k, 2l)$ for $k, l \geq 1$.

It is easy to see that $\forall i \geq 1,\ B(i, 1) \leq A(i, 2)$, and $\forall j \geq 1,\ B(1, j) \leq A(1, 2j)$.

Assume $B(k, l) \leq A(k, 2l)\, \forall k$ if $l \leq j$ and $\forall l$ if $k \leq i$. We shall show that $B(i+1, j+1) \leq A(i+1, 2(j+1))$ for $i, j \geq 1$.

$B(i+1, j+1)$ may be written as the product of the label on the edge to the rightmost child of the root, which is $2B(i+1, j)$, and the maximum product of labels in the subtree below the rightmost child of the root. The labels on the leftmost edges in this subtree are all at most $2B(i+1, j)$, thus the maximum product of labels is at most the maximum product of labels in $S_i(2B(i+1, j))$, i.e., $C(i, 2B(i+1, j))$. Hence, using (6) gives

$$B(i+1, j+1) \leq 2B(i+1, j)C(i, 2B(i+1, j)) \leq 2B(i+1, j)B(i, 4B(i+1, j)).$$

From the definition of $B(i, j)$, for $i \geq 1$, $B(i, j+1) \geq 2B(i, j)$ and hence, for $x \geq 0$, $B(i, j+x) \geq xB(i, j)$, which yields

$$2B(i+1, j)B(i, 4B(i+1, j)) \leq B(i, 6B(i+1, j)) \leq A(i, 12A(i+1, 2j))$$

where the last inequality follows from the induction hypothesis. Finally, observe that $i+1 \geq 2$ and $2j \geq 4$. It can be proved by induction that for such i, j, $12A(i+1, 2j) \leq A(i+1, 2j+1)$, and hence

$$A(i, 12A(i+1, 2j)) \leq A(i, A(i+1, 2j+1)) \leq A(i+1, 2(j+1)). \qquad \square$$

LEMMA A.2 *Proofs of inequalities* (1a)–(1e).

Proof. (1a): Obvious.

(1b): By Lemma A.1 and the choice of l, (1b) holds.

(1c): It is easy to see that the labels on edges to leaves of the tree increase by a factor of two as we move from left to right along the leaves of the tree. Thus the sum is a geometric series with the stated bound.

(1d): The sum is dominated term-by-term by the series $\sum_{h \in H} \frac{1}{c_k(h)}$, whose sum is at most 2.

(1e): Consider the sum as it is formed by going through the vertices from left to right. Since the last factor in each product increases by a factor of two and the other factors do not decrease, each term is at least twice the previous term. Since the last term is at most y the series has a sum of at most $2y$. $\square$

LEMMA A.3 *Inequality* (1f) *holds.*

Proof. Let $0 \leq i \leq k$, H_i be the set of vertices at level i of the tree and for $h \in H_i$ let $c_1(h), \ldots, c_i(h)$ be the labels on the unique path from the root to h. Then we have

$$\begin{aligned}\sum_{h\in H}\frac{1}{c_1(h)\dots c_{k-1}(h)} &= \sum_{h\in H_k}\frac{1}{c_1(h)\dots c_{k-1}(h)}\\ &= \sum_{h\in H_{k-1}}\frac{1}{c_1(h)\dots c_{k-1}(h)}c_{k-1}(h)\\ &= \cdots\\ &= \sum_{h\in H_1} 1 = l \geq \frac{I_{k+1}(n)}{3}. \qquad \square\end{aligned}$$

Acknowledgments. I am grateful to Ravi Boppana for introducing me to the work in [6] and [11], and for many helpful discussions. I am indebted to Jaikumar Radhakrishnan for the simple proof of the upper bound and for innumerable insightful comments.

REFERENCES

[1] R. Boppana, *Optimal separations between concurrent write parallel machines*, Proc. 21st ACM Symp. Th. Comput., (1989), pp. 320–326.

[2] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, *Improved deterministic parallel integer sorting*, Inform. Comput., 94 (1991), pp. 29–47.

[3] O. Berkman, J. Jaja, S. Krishnamarthy, R. Thurimella, and U. Vishkin, *Some triply-logarithmic parallel algorithms*, Proc. 31st IEEE Foundations Comp. Sci., 1990, pp. 183–185.

[4] O. Berkman and U. Vishkin, *Recursive star-tree parallel data structure*, Proc. 30th IEEE Foundations Comp. Sci., 1989, pp. 196–202.

[5] ———, *Recursive star-tree parallel data structure*, UMIACS-TR-90-40, Univ. of Maryland, 1990.

[6] A. K. Chandra, S. Fortune, and R. J. Lipton, *Unbounded fan-in circuits and associative functions*, J. Comp. Sys. Sci., 30 (1985), pp. 222–234.

[7] ———, *Lower bounds for constant depth circuits for prefix problems*, Proc. 10th Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, New York, 1983.

[8] S. Chaudhuri, *Tight bounds on the chaining problem*, Proc. 3rd ACM Symp. Par. Alg. Arch., 1991, pp. 62–70.

[9] S. Chaudhuri and J. Radhakrishnan, *The complexity of parallel prefix problems on small domains*, Proc. 33rd IEEE Foundations Comp. Sci., 1992, pp. 638–647.

[10] S. Cook, C. Dwork, and R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[11] D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson, *Superconcentrators, generalizers and generalized connectors with limited depth*, Proc. 15th ACM Symp. Th. Comput., 1983, pp. 42–51.

[12] F. E. Fich, P. Ragde, and A. Wigderson, *Simulations among concurrent-write models ofparallel computation*, Algorithmica, 3 (1988), pp. 43–51.

[13] J. Gil and L. Rudolph, *Counting and packing in parallel*, Proc. Int. Conf. Par. Proc., 1986.

[14] Y. Matias and U. Vishkin, *On integer sorting and parallel hashing*, Proc. 17th Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 443, Springer-Verlag, New York, 1990, pp. 729–743.

[15] I. Newman, P. Ragde, and A. Wigderson, *Perfect hashing, graph entropy and circuit complexity*, Proc. 5th Struct. Complex. Th., 1990, pp. 91–100.

[16] P. Ragde, *The parallel simplicity of compaction and chaining*, Proc. 17th Int. Colloq. on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 443, Springer-Verlag, New York, 1990, pp. 744–751.

[17] R. Raman, *The power of collision: randomized parallel algorithms for chaining and integer sorting*, Proc. 10th Found. Softw. Tech. Th. Comp. Sci., Lecture Notes Comput. Sci., 472 (1990), pp. 161–175.

[18] L. Rudolph and W. Steiger, *Subset selection in parallel*, Proc. 1985 Int. Conf. Par. Proc., 1985.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1266–1274, December 1994

REQUIREMENTS FOR DEADLOCK-FREE, ADAPTIVE PACKET ROUTING*

ROBERT CYPHER† AND LUIS GRAVANO‡

Abstract. This paper studies the problem of deadlock-free packet routing in parallel and distributed architectures. Three main results are presented. First, it is shown that the standard technique of ordering the buffers so that every packet always has the possibility of moving to a higher-ordered buffer is not necessary for deadlock freedom. Second, it is shown that every deadlock-free, adaptive packet routing algorithm can be restricted, by limiting the adaptivity available, to obtain an oblivious algorithm which is also deadlock-free. Third, it is shown that any packet routing algorithm for a cycle or torus network which is free of deadlock and which uses only minimal length paths must require at least three buffers in some node. This matches the known upper bound of three buffers per node for deadlock-free, minimal packet routing on cycle and torus networks.

Key words. deadlock, packet routing, store-and-forward routing, adaptive routing, networks, buffer requirements, lower bounds, parallel algorithms

AMS subject classifications. 68M10, 68Q25, 68Q22, 68Q20

1. Introduction.

This paper studies the problem of deadlock-free packet routing in parallel and distributed architectures. A wide range of packet routing algorithms with differing properties and costs has been proposed [1]–[10], [12]–[14], [16]–[21], [23], [24]. In this paper we will focus on a particularly simple and important class of routing algorithms which we will call *buffer-reservation* algorithms. A buffer-reservation algorithm consists of rules that specify to which buffers a packet may move based solely on the buffer currently holding the packet, the packet's source node, and the packet's destination node. A packet is allowed to move from its current buffer to any other buffer at any time, provided that the other buffer is empty and that the move is allowed by the routing algorithm. Buffer-reservation algorithms can be implemented efficiently in hardware because they require only local information to make routing decisions, are inherently asynchronous and therefore do not require a global clock, and do not require the creation or exchange of any special packets containing only control information. Furthermore, adaptive buffer-reservation algorithms allow packets to avoid congestion, thus permitting high throughput in the network. As a result of these advantages, buffer-reservation algorithms have been widely studied and implemented.

The primary disadvantage of buffer-reservation techniques is that they require that each node contain some minimum number of buffers. Although a great deal of research has been devoted to the problem of minimizing the storage requirements of buffer-reservation algorithms [4], [7], [8], [10], [12], [14], [16], [19], [21]–[24], very little is known in terms of lower bounds on the storage which is required by such algorithms. Our goal in this paper is to characterize the properties which these algorithms must have in order to be free of deadlock and to use these properties to prove lower bounds on storage requirements.

One well-known technique for proving freedom from deadlock is to order the buffers so that every packet always has the possibility of moving to a higher-ordered buffer [12]. Providing such an ordering of the buffers is the standard technique for proving freedom from deadlock and has been used by many researchers [4], [7], [8], [10], [12], [14], [16], [19], [21], [23], [24]. Therefore, it seems plausible that the existence of such an ordering of the buffers is a necessary condition for freedom from deadlock. In fact, in the special case of oblivious buffer-reservation algorithms, Toueg and Steiglitz have shown that the existence of such an ordering of the buffers *is* necessary for deadlock freedom [22]. However, in this paper we

*Received by the editors May 20, 1992; accepted for publication (in revised form) August 24, 1993. A preliminary version of this paper appeared in the 1992 ACM Symposium on Principles of Distributed Computing.

†IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

‡Department of Computer Science, Stanford University, Stanford, California 94305.

will present an adaptive buffer-reservation algorithm which is provably free of deadlock and for which no ordering of the buffers can be defined such that every packet always has the possibility of moving to a higher-ordered buffer. Thus, in the case of adaptive routing, the technique of ordering the buffers is sufficient but not necessary for avoiding deadlock.

On the other hand, we will prove that every deadlock-free, adaptive buffer-reservation algorithm can be restricted, by limiting the adaptivity available, to obtain an oblivious algorithm which is also deadlock-free. As a result, we will be able to use lower bounds on the storage requirements of oblivious routing algorithms to obtain lower bounds on the storage requirements of adaptive routing algorithms. In particular, we will show that any adaptive buffer-reservation algorithm for a cycle or torus network which is free of deadlock and which uses only minimal length paths must require at least three buffers in some node. This matches the known upper bound of three buffers per node for deadlock-free, minimal routing on cycle [11] and torus networks [4].

The remainder of this paper is organized as follows. Definitions and a formal description of the routing model are given in §2. Section 3 presents an example of a deadlock-free, adaptive routing algorithm in which it is impossible to order the buffers so that every packet always has the possibility of moving to a higher ordered buffer. The fact that every deadlock-free, adaptive buffer-reservation algorithm can be restricted to obtain a deadlock-free, oblivious algorithm is proven in §4. Lower bounds on the storage requirements for deadlock-free minimal buffer-reservation algorithms are given in §5. Some conclusions and open problems are presented in §6.

2. Preliminaries. We will view a routing network as being an undirected graph in which the nodes represent processors and the edges represent communication links. Each node contains a set of buffers, one of which will be called an *injection buffer*, another one of which will be called a *delivery buffer*, and the remainder of which will be called *standard buffers*. Packets can enter the routing network only by being placed in an empty injection buffer in their source node, and they can be removed from the network only when they are in the delivery buffer of their destination node. We will assume throughout that each buffer can hold exactly one packet and that the number of buffers is finite.

Given the buffer in which a packet is currently stored and the packet's source and destination nodes, a *routing algorithm* specifies a set of buffers to which the packet may be moved. More formally, the *color* of a packet is the pair (s, d) where s is the packet's source node and d is the packet's destination node. We will say that a buffer has color c if it contains a packet with color c. A routing algorithm A is a function which associates a set of buffers, called a *waiting set*, with each possible buffer and color pair (q, c). The waiting set which A associates with the pair (q, c) will be denoted $A(q, c)$. All of the buffers in a waiting set $A(q, c)$ must either be in the node which contains q or in neighboring nodes (that is, nodes that are connected by an edge to the node containing q).

If $q_2 \in A(q_1, c)$, then it is required that q_1 is not a delivery buffer and that q_2 is not an injection buffer. Furthermore, if q_2 is a delivery buffer and if $c = (s, d)$, then it is required that q_2 is in node d. Finally, and if either q_1 is an injection buffer or q_2 is a delivery buffer, then q_1 and q_2 must be in the same node. Thus injection and delivery buffers are used only for placing new packets in the network and for removing packets once they have reached their destination.[1]

The routing algorithm operates asynchronously. A packet with color c may move from a buffer q_1 to any empty buffer $q_2 \in A(q_1, c)$ at any time, and a new packet with an arbitrary

[1] It should be noted that the injection and delivery buffers are introduced only to simplify the description of the model, and that they need not be physically present in an actual routing network.

destination may be placed in an empty injection buffer at any time. Packets may be transmitted in either store-and-forward [19] or virtual cut-through [15] mode. The only requirement is that when a packet is moved from one buffer to another, it occupies both of the buffers for a finite amount of time, and after a finite amount of time the former buffer becomes empty.

A routing algorithm is *oblivious* if every waiting set contains at most one buffer, and it is *adaptive* otherwise. ("Oblivious" and "adaptive" routing algorithms have also been referred to as "static" and "dynamic" routing algorithms.) Routing algorithm A is a *restriction* of routing algorithm B if and only if for every pair (q, c), $A(q, c) \subseteq B(q, c)$, and for some pair (q, c), $A(q, c) \neq B(q, c)$. A routing algorithm is *minimal* if every packet is routed from its source node to its destination node while visiting the minimum number of nodes possible. Note that the concept of minimality is based on the number of nodes visited, rather than the number of buffers visited.

Given a routing algorithm A, a *buffer q is reachable* by a packet with color $c = (s, d)$ if and only if there exists some path $q_0, q_1, \dots, q_k$ such that q_0 is the injection buffer in node s, $q_k = q$, and for all i, $1 \leq i \leq k$, $q_i \in A(q_{i-1}, c)$. A *configuration T* is a nonempty set of buffers S such that each buffer q in S is either empty or has some color c. The set of buffers S will be called the *critical set* of the configuration T. Given a configuration T with critical set S and given any buffer $q \in S$, the notation $T(q)$ will denote q's color in configuration T (or the value "empty" if it does not contain a packet). A *configuration is reachable* if and only if it is possible to start with an empty network and to route packets so as to obtain the configuration.

A *deadlock configuration* for a routing algorithm A is a configuration with a critical set S such that none of the buffers in S is a delivery buffer, none of the buffers in S is empty, and for each buffer q in S, q has color c where $A(q, c) \subseteq S$. A routing algorithm is *deadlock-free* if and only if it has no reachable deadlock configuration. It is straightforward to verify that this definition of deadlock-freedom does in fact correspond to the impossibility of obtaining deadlock when using the given routing algorithm. Finally, given any two configurations T' and T'' with critical sets S' and S'', respectively, $T' \oplus T''$ will denote the configuration T with critical set $S = S' \cup S''$ in which for each buffer $q \in S'$, $T(q) = T'(q)$, and for each buffer $q \in S'' \setminus S'$, $T(q) = T''(q)$. Thus $T' \oplus T''$ is obtained by taking configuration T' and adding to it all of those buffers in T'' which are not also in T'.

3. Deadlock freedom without ordering buffers. In this section we will show that the standard technique of ordering the buffers so that every packet always has the possibility of moving to a higher ordered buffer is not necessary for the prevention of deadlock in buffer-reservation algorithms. In particular, we will give a simple example of an adaptive buffer-reservation algorithm which is provably free of deadlock and has no such ordering of the buffers.

The example is routing algorithm A shown in Fig. 1, in which each circle represents a buffer and each arc represents a possible move between buffers. There are three injection buffers labeled I_1, I_2, and I_3, and three delivery buffers labeled D_1, D_2, and D_3. In addition, there are six standard buffers labeled X_1, X_2, X_3, Y_1, Y_2, and Y_3. We will consider only three colors of packets, namely C_1, C_2, and C_3, where packets with color C_i, $1 \leq i \leq 3$, are injected in buffer I_i and delivered from buffer D_i. The label associated with each arc specifies which color packets are allowed to make the given move between buffers. For example, $A(I_1, C_1) = \{X_1\}$, $A(X_1, C_1) = \{X_2, Y_1\}$, and $A(X_1, C_2) = \{X_3\}$. Of course, a complete routing algorithm would provide routes for packets with other colors. In particular, note that for each other color (s, d), it would be possible to add a dedicated path of buffers from the injection buffer in s to the delivery buffer in d. These dedicated paths would not change the deadlock or buffer ordering properties of the example, and they would allow packets to be routed from each injection buffer to each delivery buffer. Finally, note that routing algorithm A does not include an assignment of buffers to nodes, but the creation of such an assignment is straightforward.

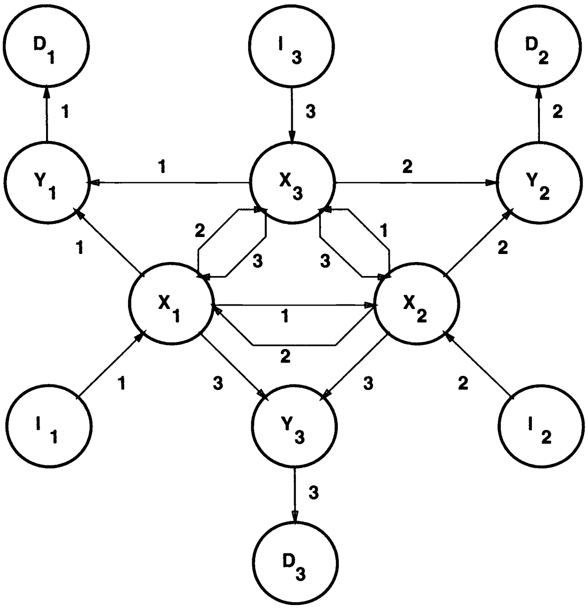


FIG. 1. *A deadlock-free, adaptive routing algorithm A for which the technique of ordering the buffers cannot be used to prove freedom from deadlock.*

LEMMA 3.1. *The routing algorithm A shown in Fig.* 1 *is free of deadlock.*

Proof. Assume for the sake of contradiction that deadlock is possible, in which case there must be some reachable deadlock configuration with a nonempty critical set S that does not contain any delivery buffers. Note that Y_1, Y_2, and Y_3 cannot appear in S because they are only reachable by packets which are able to move directly to a delivery buffer. Also, note that if injection buffer I_i, $1 \leq i \leq 3$ is in S, then buffer X_i must also be in S. Therefore, at least one of the buffers X_i must be in S. Because none of the buffers Y_i is in S, it follows that if X_1 is in S it must have color C_2 in the deadlock configuration; if X_2 is in S it must have color C_1 in the deadlock configuration; if X_3 is in S it must have color C_3 in the deadlock configuration. Note that X_3 must be in S, because otherwise either X_1 or X_2 must be in S, and $X_3 \in A(X_1, C_2)$ and $X_3 \in A(X_2, C_1)$. Because $X_1 \in A(X_3, C_3)$ and $X_2 \in A(X_3, C_3)$, both X_1 and X_2 must be in S. Therefore, the deadlock configuration must include a C_2 packet in X_1 and a C_1 packet in X_2. However, it is impossible to simultaneously route a C_2 packet to X_1 and a C_1 packet to X_2, so the deadlock configuration is not reachable, which is a contradiction. □

LEMMA 3.2. *There is no ordering of the buffers shown in Fig.* 1 *such that every packet always has the possibility of moving to a higher-ordered buffer.*

Proof. Assume for the sake of contradiction that such an ordering is possible. Note that X_1 is reachable by a packet with color C_2 and $A(X_1, C_2) = \{X_3\}$ and that X_2 is reachable by a packet with color C_1 and $A(X_2, C_1) = \{X_3\}$. Therefore, buffer X_3 must be higher ordered than both X_1 and X_2. However, X_3 is reachable by a packet with color C_3 and $A(X_3, C_3) = \{X_1, X_2\}$, so either X_1 or X_2 (or both) must be higher ordered than X_3, which is a contradiction. □

Combining the two previous lemmas yields the following theorem.

THEOREM 3.3. *There exists an adaptive routing algorithm which is free of deadlock, and for which there is no ordering of the buffers such that every packet always has the possibility of moving to a higher-ordered buffer.*

4. Restrictions of adaptive routing algorithms. In this section we will show that every deadlock-free, adaptive packet routing algorithm can be restricted to obtain an oblivious algorithm which is also deadlock-free. The proof will depend on the following lemma.

LEMMA 4.1. *Let A be any deadlock-free, adaptive routing algorithm, let q_1 be any buffer, and let c_1 be any color such that $|A(q_1, c_1)| \geq 2$. Let q_2 be any buffer such that $q_2 \in A(q_1, c_1)$, and let B be the restriction of A obtained by removing q_2 from the waiting set associated with (q_1, c_1). If B is subject to deadlock, then there must exist some reachable deadlock configuration for B in which buffer q_1 contains a packet with color c_1.*

Proof. Because B is subject to deadlock, there must exist some configuration T which is a reachable deadlock configuration for B. Because B is a restriction of A, it follows that configuration T is also reachable by A. However, A is deadlock-free, so T must not be a deadlock configuration for A. Therefore, buffer q_1 must have color c_1 in configuration T. □

THEOREM 4.2. *Given any adaptive, deadlock-free routing algorithm A, there exists an oblivious, deadlock-free routing algorithm B which is a restriction of A.*

Proof. Assume for the sake of contradiction that the claim is false. Then there must exist some adaptive, deadlock-free routing algorithm A such that every routing algorithm A' which is a restriction of A is subject to deadlock. Let A be such a deadlock-free routing algorithm, q_1 be any buffer, and c_1 be any color such that $|A(q_1, c_1)| \geq 2$. Let q_2 and q_3 be any distinct buffers such that $q_2 \in A(q_1, c_1)$ and $q_3 \in A(q_1, c_1)$, let A' be the restriction of A obtained by removing q_2 from the waiting set associated with (q_1, c_1), and let A'' be the restriction of A obtained by removing q_3 from the waiting set associated with (q_1, c_1). It follows from Lemma 4.1 that there exists a configuration T' (similarly, T'') which is a reachable deadlock configuration for A' (similarly, A'') and in which buffer q_1 contains a packet with color c_1. Let $T = T' \oplus T''$ and let S be the critical set of T. Let R be the configuration which also has critical set S but in which all of the buffers are empty. Note that the following properties hold.

Property 1: The set S is the critical set of both configuration T and configuration R.

Property 2: Configuration R is a reachable configuration for A.

Property 3: Every buffer q in R is either empty or has a color c such that $A(q, c) \subseteq S$.

Property 4: Every buffer q in T has a color c such that q is reachable by a packet with color c.

Property 5: Configuration T is a deadlock configuration for A.

Also note that Property 5 implies that S does not contain any delivery buffers.

We will define an algorithm for transforming R and T while maintaining Properties 1–5. The algorithm will repeatedly add packets to empty buffers in R until none of the buffers in R is empty. At this point R will be a reachable deadlock configuration, which will be the desired contradiction.

The algorithm for transforming R and T consists of repeatedly performing the following subroutine until R contains no empty buffers. The subroutine consists of two halves. In the first half, we will pick an arbitrary buffer in R which is empty and we will fill it. We will then extend both T and R so that the selected buffer is filled in R and so that Properties 1, 2, and 4 hold. In the second half, we will extend both T and R still further so that Properties 1–5 hold. The two halves of the subroutine operate as follows.

First, select an arbitrary buffer q which is empty in R. Let $c = T(q)$ and note that $A(q, c) \subseteq S$ (from Property 5). Because buffer q is reachable by some packet p with color c (from Property 4), there must exist a simple path from p's injection buffer to buffer q. Define the configuration P in which the critical set consists of all of those buffers that appear in this simple path, and in which all of the buffers in the critical set contain a packet with color c. Transform R to become the configuration obtained by adding P and R (that is, perform the assignment $R \leftarrow P \oplus R$), transform T to become the configuration obtained by adding P and T (that is, perform the assignment $T \leftarrow P \oplus T$), and let S be the critical set of the transformed configurations R and T. This completes the first half of the subroutine.

At this point, Property 1 clearly holds. Also, Property 2 holds because it is possible to first route packets with the desired colors to all of the nonempty buffers in R which are not in P and to then fill the buffers in P with packets with color c. Furthermore, Property 4 holds because all of the buffers in P are reachable by packets with color c. Finally, note that S does not contain any delivery buffers because configuration P does not contain any delivery buffers. However, Properties 3 and 5 may not hold, because it is possible that some of the packets in P have waiting sets that include buffers which are not in S.

The second half of the subroutine repeatedly adds buffers to T and R until Properties 1–5 hold. First, select an arbitrary buffer q' in the simple path described above such that $A(q', c) \not\subseteq S$ (if such a buffer exists). Let q'' be the successor of q' in the simple path described above (note that q'' must exist if q' exists, because $A(q, c) \subseteq S$ so $q' \neq q$). Let A' be the restriction of A obtained by removing q'' from the waiting set associated with (q', c). It follows from Lemma 4.1 that there exists a configuration D which is a reachable deadlock configuration for A' and in which buffer q' contains a packet with color c. Let D' be the configuration with the same critical set as D but in which all of the buffers are empty. Transform R to become the configuration obtained by adding R and D' (that is, perform the assignment $R \leftarrow R \oplus D'$), transform T to become the configuration obtained by adding T and D (that is, perform the assignment $T \leftarrow T \oplus D$), and let S be the critical set of the transformed configurations R and T. Repeat this procedure of selecting a buffer q' in the simple path such that $A(q', c) \not\subseteq S$ and transforming R, T, and S until no such buffer q' exists. When no such buffer q' exists, the second half of the subroutine is completed. At this point return from the subroutine.

It is clear that the transformations performed in the second half of the subroutine maintain Properties 1, 2, and 4. Furthermore, upon returning from the subroutine Property 3 must hold because for each buffer q' in the simple path described above, $A(q', c) \subseteq S$. Finally, upon returning from the subroutine Property 5 must hold because S does not contain any delivery buffers and for each buffer q in T with color c, $A(q, c) \subseteq S$. Therefore, all of Properties 1–5 hold upon returning from the subroutine.

Note that any buffer in R which was nonempty before calling the subroutine will again be nonempty after calling the subroutine. Finally, note that following the call to the subroutine, R contains at least one additional nonempty buffer. Because the number of buffers is finite, this procedure must terminate, at which point R is both reachable by A (from Property 2) and a deadlock configuration for A (from Property 3 and from the fact that R contains neither empty buffers nor delivery buffers), which is a contradiction. □

5. Minimal routing in cycle and torus networks. In this section we will prove lower bounds on the number of buffers per node that are required for deadlock-free, minimal routing in cycle and torus networks. Our approach will be to first prove a lower bound on the buffer requirements of deadlock-free, minimal, oblivious routing algorithms for cycle networks. We will then use this lower bound, along with Theorem 4.2 and the fact that a torus network can be decomposed into disjoint cycles, to obtain a lower bound on the buffer requirements of deadlock-free, minimal routing algorithms for both cycle and torus networks. Of course, it should be recalled that we are concerned solely with buffer-reservation algorithms, and that other types of deadlock-free routing algorithms for cycle networks have been created [10],

[20], [23]. However, these other types of routing algorithms have fundamentally different properties. For example, buffer insertion rings [20] can provide high throughput, but they are poorly suited to the creation of minimal routing algorithms for torus networks.

LEMMA 5.1. *Let routing algorithm A be any deadlock-free, minimal, and oblivious routing algorithm for a cycle network with n nodes. The cycle network must contain at least* $3n - 30$ *standard buffers.*

Proof. Because A is deadlock-free and oblivious, it follows that there exists an ordering of the buffers such that every packet visits the buffers in ascending order [22]. Let $k = \lfloor n/2 \rfloor - 1$ (so either $n = 2k + 2$ or $n = 2k + 3$). We will say that a packet is routed in the *clockwise* direction if it visits buffers in nodes of the form i, $(i + 1) \bmod n$, $(i + 2) \bmod n, \ldots, j$, and in the *counterclockwise* direction otherwise. Note that for each node i, $0 \le i < n$, algorithm A routes packets from node i to node $(i + k) \bmod n$ in the clockwise direction. Therefore, for each node i, $0 \le i < n$, there must exist an ascending sequence of standard buffers $s_{i,i}\ , s_{i,(i+1)\mathrm{mod}n}\ , \ \ldots\ , s_{i,(i+k)\mathrm{mod}n}$, where each buffer of the form $s_{i,j}$ is located in node j (for example, let $s_{i,j}$ be the highest ordered standard buffer in node j which is visited by a packet with source node i and destination node $(i + k) \bmod n$). For each i, $0 \le i < n$, let $S_i = s_{i,i}\ , s_{i,(i+1)\mathrm{mod}n}\ , \ \ldots\ , s_{i,(i+k)\mathrm{mod}n}$ denote the ascending sequence of standard buffers beginning in node i. Let $h = n - 1 - k$, note that $S_h = s_{h,h}\ , s_{h,h+1}\ , \ \ldots\ , s_{h,n-1}$, and note that S_0 and S_h are disjoint.

We will say that a sequence of buffers is a *clockwise increasing* (similarly, *counterclockwise increasing*) sequence if when the buffers are visited in ascending order, the nodes containing the buffers are visited in clockwise (counterclockwise) order. We will show that there must exist at most three mutually disjoint clockwise increasing sequences of buffers, the total length of which is at least $n + k$. There are two cases, which follow.

Case 1. There exists a clockwise increasing sequence of standard buffers $X = x_0\ , x_1\ , \ \ldots\ , x_{n-1}$ such that for each i, $0 \le i < n$, x_i is located in node i. In this case, we have two subcases.

Case 1(a). There exists an a, $0 \le a \le n - 1$, such that S_a and X are disjoint. In this case, the two disjoint clockwise increasing sequences are S_a and X, and their total length is $n + k + 1$.

Case 1(b). For each i, $0 \le i \le n - 1$, S_i and X intersect. In this case, let a be the largest value of i, $0 \le i \le n - 1$, such that there exists a value $i' \ge i$ where $s_{i,i'} = x_{i'}$. Let a' be any value such that $a' \ge a$ and $s_{a,a'} = x_{a'}$. Let $b = (a + 1) \bmod n$ and let b' be any value such that $s_{b,b'} = x_{b'}$. Note that $a' \ge a \ge k \ge b'$. Let Y be the sequence

$$s_{b,b}\ , \ \ldots\ , s_{b,b'-1}\ , x_{b'}\ , \ \ldots\ , x_{a'}\ , s_{a,a'+1}\ , \ \ldots\ , s_{a,(a+k)\mathrm{mod}n}.$$

The sequence Y is clockwise increasing and has length $n + k$.

Case 2. There does not exist such a sequence X. In this case, we have two subcases.

Case 2(a). There exists an a, $0 \le a \le h$, such that S_a and S_0 are disjoint and such that S_a and S_h are disjoint. In this case, the three disjoint clockwise increasing sequences are S_0, S_a, and S_h, and their total length is $3k + 3 \ge n + k$.

Case 2(b). For each i, $0 \le i \le h$, either S_i and S_0 intersect or S_i and S_h intersect, but not both. In this case, let a be the largest value of i in the range $0 \le i \le h$ such that S_i and S_0 intersect. Let a' be any value such that $s_{a,a'} = s_{0,a'}$. Let $b = a + 1$ and let b' be any value such that $s_{b,b'} = s_{h,b'}$ (note that such a b' must exist because of the definition of a and the fact that S_h does not intersect S_0). Let Y be the sequence

$$s_{0,0}\ , \ \ldots\ , s_{0,a'-1}\ , s_{a,a'}\ , \ \ldots\ , s_{a,a+k}$$

and let Z be the sequence

$$s_{b,b}\ ,\ \dots\ ,\ s_{b,b'}\ ,\ s_{h,b'+1}\ ,\ \dots\ ,\ s_{h,n-1}.$$

Note that Y and Z are clockwise increasing sequences and that they must be disjoint (because otherwise there would exist a clockwise increasing sequence X spanning all of the nodes). Also, note that the length of Y is $a + k + 1$ and the length of Z is $n - a - 1$, so their total length is $n + k$.

Thus, in any case there must exist at most three mutually disjoint clockwise increasing sequences of buffers, the total length of which is exactly $n+k$. An analogous argument can be used to show that there must exist at most three mutually disjoint counterclockwise increasing sequences of buffers, the total length of which is exactly $n + k$.

Now consider an arbitrary clockwise increasing sequence of buffers Y and an arbitrary counterclockwise increasing sequence of buffers Z. Assume that Y and Z intersect at some buffer a and again at some higher ordered buffer b, let Y' be the subsequence of Y starting at a and ending immediately before b, and let Z' be the subsequence of Z starting at a and ending immediately before b. The total length of Y' and Z' must be at least n, because every node other than the node containing b must contain at least one buffer from either Y' or Z', and the node containing a must contain at least one buffer from both Y' and Z'. Now assume that Y and Z intersect x times and for $1 \leq i < x$ let Y_i denote the subsequence of Y starting at their ith point of intersection and ending immediately before their i + 1st point of intersection, let Y_x denote the subsequence of Y of length 1 consisting of their xth point of intersection, and let Z_i subsequences be defined analogously. Then for $1 \leq i < x$, the total length of Y_i and Z_i must be at least n, and the total length of Y_x and Z_x must be 2, so the total length of Y and Z must be at least $(x - 1)n + 2$.

As a result, a clockwise increasing sequence of buffers of length at most $n + k$ and a counterclockwise increasing sequence of buffers of length at most $n+k$ can intersect in at most three buffers. Therefore, it follows that the entire collection of clockwise increasing sequences and counterclockwise increasing sequences contains at least $(n + k) + (n + k) - (3 * 9) = 2n + 2k - 27 \geq 3n - 30$ distinct buffers. □

THEOREM 5.2. *Let routing algorithm A be any deadlock-free, minimal routing algorithm for a cycle network with* 31 *or more nodes or for a torus network in which at least one of the dimensions is of length* 31 *or greater. The cycle or torus network must contain at least one node which has three or more standard buffers.*

Proof. The claim for a cycle network follows immediately from Theorem 4.2 and Lemma 5.1. The claim for a torus network follows from Theorem 4.2, Lemma 5.1, and the observation that a torus can be decomposed into disjoint cycles such that all minimal length paths between pairs of nodes within a cycle lie within the cycle. □

6. Conclusions and open problems. This paper has examined the characteristics which an adaptive packet routing algorithm must have in order to be free of deadlock. In particular, we have shown that an adaptive packet routing algorithm may be free of deadlock even though it is impossible to order the buffers so that every packet always has the possibility of moving to a higher-ordered buffer. On the other hand, we have shown that every deadlock-free, adaptive buffer-reservation algorithm can be restricted to obtain a deadlock-free oblivious algorithm. We have also used this fact to show lower bounds on the storage requirements of minimal, deadlock-free routing algorithms for cycle and torus networks.

These results suggest a number of natural open problems. It would be interesting to obtain lower bounds on the storage requirements of minimal, deadlock-free routing algorithms for different topologies, including hypercube-derivative networks such as de Bruijn and butterfly networks. It would also be interesting to see whether or not any networks exist for which these lower bounds on storage requirements are nonconstant. Finally, the problem of characterizing the properties of deadlock-free wormhole routing algorithms remains open.

REFERENCES

[1] B. Awerbuch, S. Kutten, and D. Peleg, *Efficient deadlock-free routing*, in Proc. ACM Symposium on Principles of Distributed Computing, 1991, pp. 177–188.

[2] P. E. Berman, L. Gravano, G. D. Pifarré, and J. L. C. Sanz, *Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks*, in Proc. ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 3–12.

[3] A. A. Chien and J. H. Kim, *Planar-adaptive routing: Low-cost adaptive networks for multiprocessors*, in Proc. 19th Intl. ACM Symposium on Computer Architecture, ACM, New York, 1992, pp. 268–277.

[4] R. Cypher and L. Gravano, *Adaptive, deadlock-free packet routing in torus networks with minimal storage*, in Proc. Intl. Conf. on Parallel Processing, vol. 3, pp. 204–211, 1992.

[5] W. J. Dally and C. Seitz, *Deadlock-free message routing in multiprocessor interconnection networks*, IEEE Transactions on Computers, 36 (1987), pp. 547–553.

[6] J. Duato, *Deadlock-free adaptive routing algorithms for multicomputers: evaluation of a new algorithm*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, IEEE, Washington, D.C., pp. 840–847, 1991.

[7] S. A. Felperin, H. Laffitte, G. Buranits, and J. L. C. Sanz, *Deadlock-free minimal packet routing in the torus network*, Tech. Rep. TR:91-22, IBM Computer Research and Advanced Applications Group, Buenos Aires, Argentina, 1991.

[8] B. Gavish, P. M. Merlin, and P. J. Schweitzer, *Minimal buffer requirements for avoiding store-and-forward deadlock*, Tech. Rep. RC 6672, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1977.

[9] C. J. Glass and L. M. Ni, *The turn model for adaptive routing*, in Proc. 19th Intl. Symposium on Computer Architecture, pp. 278–287, 1992.

[10] I. S. Gopal, *Prevention of store-and-forward deadlock in computer networks*, IEEE Trans. Comm., 33 (1985), pp. 1258–1264.

[11] L. Gravano, G. D. Pifarré, S. A. Felperin, and J. L. C. Sanz, *Adaptive deadlock-free worm-hole routing with all minimal paths*, Tech. Rep. TR:91-21, IBM Computer Research and Advanced Applications Group, Buenos Aires, Argentina, 1991.

[12] K. D. Günther, *Prevention of deadlocks in packet-switched data transport systems*, IEEE Trans. Comm., 29 (1981), pp. 512–524.

[13] P. A. J. Hilbers and J. J. Lukkien, *Deadlock-free message routing in multicomputer networks*, Distributed Computing, 3 (1989), pp. 178–186.

[14] C. R. Jesshope, P. R. Miller, and J. T. Yantchev, *High performance communications in processor networks*, in Proc. 16th Intl. Symposium on Computer Architecture, pp. 150–157, 1989.

[15] P. Kermani and L. Kleinrock, *Virtual cut-through: a new computer communication switching technique*, Comput. Networks, 3 (1979), pp. 267–286.

[16] S. Konstantinidou, *Adaptive, minimal routing in hypercubes*, in Proc. 6th. MIT Conference on Advanced Research in VLSI, pp. 139–153, 1990.

[17] S. Konstantinidou and L. Snyder, *The Chaos router: A practical application of randomization in network routing*, in Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 21–30, 1990.

[18] Y. Mansour and B. Patt-shamir, *Greedy packet scheduling on shortest paths*, in Proc. ACM Symposium on Principles of Distributed Computing, pp. 165–175, 1991.

[19] P. M. Merlin and P. J. Schweitzer, *Deadlock avoidance in store-and-forward networks. 1: Store-and-forward deadlock*, IEEE Trans. Comm., 28 (1980), pp. 345–354.

[20] Y. Ofek and M. Yung, *Principles for high speed network control: loss-less and deadlock-freeness, self-routing and a single buffer per link*, in Proc. ACM Symposium on Principles of Distributed Computing, pp. 161–175, 1990.

[21] G. D. Pifarré, L. Gravano, S. A. Felperin, and J. L. C. Sanz, *Fully-adaptive minimal deadlock-free packet routing in hypercubes, meshes, and other networks*, in Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures, pp. 278–290, 1991.

[22] S. Toueg and K. Steiglitz, *Some complexity results in the design of deadlock-free packet switching networks*, SIAM J. Comput., 10 (1981), pp. 702–712.

[23] S. Toueg and J. D. Ullman, *Deadlock-free packet switching networks*, SIAM J. Comput., 10 (1981), pp. 594–611.

[24] J. Yantchev and C. R. Jesshope, *Adaptive, low latency, deadlock-free packet routing for networks of processors*, IEE Proc., Pt. E, 136 (1989), pp. 178–186.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1275–1282, December 1994

011

ON LANGUAGES REDUCIBLE TO ALGORITHMICALLY RANDOM LANGUAGES*

RONALD V. BOOK†

Abstract. In this paper languages "bounded reducible" to *algorithmically random languages* are studied; these are the languages whose characteristic sequences are algorithmically random (as defined by Martin-Löf [*Inform. and Control*, 9 (1966), pp. 602–619]); here RAND denotes the class of algorithmically random languages. The reducibilities $\leq^{\mathcal{R}}$ are very general but are defined so that if $A \in \mathcal{R}(B)$, then there is a machine M with the properties that $L(M, B) = A$ and every computation of M relative to any oracle halts.

Book, Lutz, and Wagner [*Math. Systems Theory*, 27 (1994), pp. 201–209] studied ALMOST-$\mathcal{R}$, defined to be $\{A \mid$ for almost every B, $A \leq_{\mathcal{R}} B\}$. They showed that ALMOST-$\mathcal{R} = \mathcal{R}(\text{RAND}) \cap \text{REC}$, where REC denotes the class of recursive languages, so that ALMOST-$\mathcal{R}$ is the "recursive part" of $\mathcal{R}(\text{RAND})$. In this paper this characterization is strengthened by showing that for *every* $B \in \text{RAND}$, ALMOST-$\mathcal{R} = \mathcal{R}(B) \cap \text{REC}$. A pair (A, B) of languages is an *independent pair* of algorithmically random languages if $A \oplus B \in \text{RAND}$. In this paper it is shown that for every $\mathcal{R}$ and for every independent pair (A, B), ALMOST-$\mathcal{R} = \mathcal{R}(A) \cap \mathcal{R}(B)$.

Key words. reducibilities, algorithmically random oracles, complexity classes, characterization theorems, independent pairs

AMS subject classifications. 68Q15, 03D15

1. Introduction.

Within computational complexity theory there has been a growing interest in probabilistic algorithms (or probabilistic machines) and randomness. Probabilistic machines incorporate randomness into their underlying logic, so that randomization is part of the internal calculation. Just as in the case of nondeterministic machines, there are examples where probabilistic algorithms can solve problems much faster than is known to be possible by deterministic algorithms. Hence, these concepts have received a great deal of attention.

In structural complexity theory "random oracles" have been studied when using the method of reducibilities to compare complexity classes. Many of the results are stated in terms of a relationship being true for "almost every oracle." (See [Amb86], [BG81], [Cai89], [NW88].) The results given there suggest that one might consider a (nicely behaved) reducibility or a relativization of a complexity class, say $\mathcal{R}$, and define a class ALMOST-$\mathcal{R}$ as $\{B \mid$ for almost every language A, $B \in \mathcal{R}(A)\}$. For example, if $\mathcal{R}$ is the reducibility $\leq^{\text{P}}_{\text{T}}$, then ALMOST-$\mathcal{R}$ is the class BPP, and if $\mathcal{R}$ is the reducibility $\leq^{\text{NP}}_{\text{T}}$, then ALMOST-$\mathcal{R}$ is the class AM. Properties of ALMOST-$\mathcal{R}$ for appropriate reducibilities $\mathcal{R}$ were studied by Book, Lutz, and Wagner [BLW94].

In this context one must consider the class of "algorithmically random" languages of Martin-Löf [Mar66]. As Ko [Ko92], [Ko86] has observed, the notion of algorithmic randomness is the strongest definition of random languages (or sequences) that is widely accepted as *the* definition of randomness. Book, Lutz, and Wagner showed that for appropriate reducibilities $\mathcal{R}$, a recursive language A is in ALMOST-$\mathcal{R}$ if and only if there is an algorithmically random language B such that $A \in \mathcal{R}(B)$, so that ALMOST-$\mathcal{R} = \mathcal{R}(\text{RAND}) \cap \text{REC}$, where REC denotes the class of recursive languages, and ALMOST-$\mathcal{R}$ represents the recursive part of $\mathcal{R}(A)$.

The principal results of the present paper are two additional characterizations of ALMOST-$\mathcal{R}$, each of which is strong.

*Received by the editors January 25, 1993; accepted for publication (in revised form) August 30, 1993. The preparation of this paper was supported in part by National Science Foundation grants CCR-8913584 and CCR-9302057, and by the Japan Society for the Promotion of Science. Some of these results were presented at ISAAC-93, the Third International Symposium on Algorithms and Complexity, held in December 1993 in Hong Kong.

†Department of Mathematics, University of California, Santa Barbara, California 93106 (book@math.ucsb.edu).

The random oracle characterization. For every $B \in$ RAND, ALMOST-$\mathcal{R} = \mathcal{R}(B) \cap$ REC (Theorem 3.5).

The independent pair characterization. For every pair (A, B) of languages such that $A \oplus B \in$ RAND, ALMOST-$\mathcal{R} = \mathcal{R}(A) \cap \mathcal{R}(B)$ (Theorem 4.2).

2. Preliminaries. For the most part our notation is standard, following that used by Balcázar, Díaz, and Gabarró [BDG88], [BDG90]. We assume that the reader is familiar with the standard recursive reducibilities and the variants obtained by imposing resource bounds such as time or space on the algorithms that compute these reducibilities.

A *word* (string) is an element of $\{0, 1\}^*$. The length of a word $w \in \{0, 1\}^*$ is denoted $|w|$. For a set A of strings and an integer $n > 0$, let $A^{\leq n} = \{x \in A \mid |x| \leq n\}$.

The power set of a set A is denoted by $\mathcal{P}(A)$.

Let c_A be the characteristic function of A. The *characteristic sequence* χ_A of a language A is the infinite sequence $c_A(x_0)c_A(x_1)c_A(x_2)\ldots$, where $\{x_0, x_1, x_2, \ldots\} = \{0, 1\}^*$ in lexicographical order. We freely identify a language with its characteristic sequence and the class Ω of all languages on the fixed finite alphabet $\{0, 1\}$ with the set $\{0, 1\}^\omega$ of all such infinite sequences; the usage is based on context so that there will be no ambiguity on the part of the reader.

If X is a set of strings (i.e., a language) and $\mathbf{C}$ is a set of sequences (i.e., a class of languages), then $X \cdot \mathbf{C}$ denotes the set $\{w\xi \mid w \in X, \ \xi \in \mathbf{C}\}$.

For each string w, $\mathbf{C}_w = \{w\} \cdot \{0, 1\}^\omega$ is the *basic open set* defined by w. An *open set* is a (finite or infinite) union of basic open sets, i.e., a set $X \cdot \{0, 1\}^\omega$ where $X \subseteq \{0, 1\}^*$. (This definition gives the usual product topology, also known as the Cantor topology, on $\{0, 1\}^\omega$.) A *closed set* is the complement of an open set. A class of languages is *recursively open* if it is of the form $X \cdot \{0, 1\}^\omega$ for some recursively enumerable set $X \subseteq \{0, 1\}^*$. A class of languages is *recursively closed* if it is the complement of some recursively open set.

We assume an effective enumeration of the recursively enumerable languages as $W_1, W_2, \ldots$.

For a class $\mathbf{C}$ of languages we write Prob[$\mathbf{C}$] for the probability that $A \in \mathbf{C}$ when A is chosen by a random experiment in which an independent toss of a fair coin is used to decide whether a string is in A. This probability is defined whenever $\mathbf{C}$ is measurable in the usual product topology of $\{0, 1\}^*$. In particular, if $\mathbf{C}$ is a countable union or intersection of (recursively) open or closed sets, then $\mathbf{C}$ is measurable and so Prob[$\mathbf{C}$] is defined. Note that there are only countably many recursively open sets, so every intersection of recursively open sets is a countable intersection of such sets, and hence is measurable; similarly every union of recursively closed sets is measurable. Notice that the statement "almost every B is in $\mathbf{C}$" corresponds to the more formal statement "Prob$[\{B \mid B \in \mathbf{C}\}] = 1$."

A class $\mathbf{C}$ is *closed under finite variation* if $A \in \mathbf{C}$ holds whenever $B \in \mathbf{C}$ and A and B have finite symmetric difference. A class $\mathbf{C}$ is *closed under finite translation* if for all $y \in \{0, 1\}^*$ and all $A \subseteq \{0, 1\}^\omega$, $\{y\} \cdot A \in \mathbf{C}$ implies $A \in \mathbf{C}$.

The Kolmogorov 0–1 law says that every measurable class $\mathbf{C} \subseteq \{0, 1\}^\omega$ that is closed under finite variation has either measure 0 or measure 1.

3. The random oracle characterization. The definition of an "algorithmically random" language is due to Martin-Löf [Mar66]. A class $\mathbf{C}$ is called a *constructive null set* if there is a total recursive function g with the properties that for every k we have the following.

(i) $\mathbf{C} \subseteq W_{g(k)} \cdot \{0, 1\}^\omega$, and

(ii) Prob$[W_{g(k)} \cdot \{0, 1\}^\omega] \leq 2^{-k}$.

Hence, every constructive null set has measure 0. Let NULL be the union of all constructive null sets, and let RAND $=_{df}$ $\{0, 1\}^\omega$ − NULL be the class of *algorithmically random* languages. Since NULL is a countable union of measure 0 sets, we have Prob[NULL] $= 0$ and, consequently, Prob[RAND] $= 1$.

No recursively enumerable language is in RAND.

Since we are concerned with the use of oracles, we consider complexity classes that can be specified so as to "relativize." But we want to do this in a more general setting than reducibilities computed in polynomial time, and so we introduce a few definitions.

Assume a fixed enumeration $M_0, M_1, M_2, \ldots$ of deterministic oracle Turing machines.

A *relativized class* is a function $\mathbf{C} : \mathcal{P}(\{0,1\}^*) \longrightarrow \mathcal{P}(\mathcal{P}(\{0,1\}^*))$. A *recursive presentation* of a relativized class $\mathbf{C}$ of languages is a total recursive function $f : \mathbf{N} \longrightarrow \mathbf{N}$ such that for every language A and every $i \geq 0$, $M_{f(i)}(A)$ halts on every computation and $\mathbf{C}(A) = \{L(M_{f(i)}(A)) \mid i \in \mathbf{N}\}$. A relativized class is *recursively presentable* if it has a recursive presentation.

Notice that if for every A and every i, $M_{f(i)}(A)$ halts on every computation, then $M_{f(i)}()$ has a running time that is bounded above by a recursive function.

A *reducibility* is a relativized class. A *bounded reducibility* is a relativized class that is recursively presentable. If $\mathcal{R}$ is a reducibility, then we use the notation $A \leq^{\mathcal{R}} B$ to indicate that $A \in \mathcal{R}(B)$. In addition we write $\mathcal{R}^{-1}(A)$ for $\{B \mid A \leq^{\mathcal{R}} B\}$. Typical bounded reducibilities include $\leq_m^{\mathrm{P}}, \leq_{btt}^{\mathrm{P}}, \leq_T^{\mathrm{P}}, \leq_T^{\mathrm{NP}}, \leq_T^{\mathrm{SN}}, \leq_m^{\mathrm{logspace}}$, etc. The relations $\leq_m$ and $\leq_T$ are reducibilities that are not bounded.

If $\mathcal{R}$ is a reducibility and $\mathbf{C}$ is a set of languages, write $\mathcal{R}(\mathbf{C})$ for $\bigcup_{A \in \mathbf{C}} \mathcal{R}(A)$.

A reducibility $\mathcal{R}$ will be called *appropriate* if (i) it is bounded, (ii) for any language A, $\mathcal{R}(A)$ is closed under finite variation, and (iii) for any language A, $\mathcal{R}^{-1}(A)$ is closed under finite translation and under finite variation.

The reader should note that the reducibilities commonly used in structural complexity meet the conditions for being appropriate.

There are characterizations of complexity classes in terms of RAND. This follows from the next fact.

PROPOSITION 3.1 [BLW94]. *If $\mathcal{R}$ is an appropriate reducibility, then the inverse image $\mathcal{R}^{-1}(B)$ of a recursive language B is a union of recursively closed sets.*

For each reducibility $\mathcal{R}$, define $\mathsf{ALMOST}\text{-}\mathcal{R} = \{A \mid \mathrm{Prob}[\{B \mid A \in \mathcal{R}(B)\}] = 1\}$. Let REC denote the class of recursive languages.

PROPOSITION 3.2 [BLW94]. *If $\mathcal{R}$ is an appropriate reducibility, then* $\mathsf{ALMOST}\text{-}\mathcal{R} = \mathcal{R}(\mathrm{RAND}) \cap \mathrm{REC}$.

Simple examples of characterizations of complexity classes given by Proposition 3.2 are the following:

(a) $\mathrm{P} = \mathrm{P}_{btt}(\mathrm{RAND}) \cap \mathrm{REC} = \mathsf{ALMOST}\text{-}\mathrm{P}_{btt}$ [Amb86], [BG81];

(b) $\mathrm{BPP} = \mathrm{P}_T(\mathrm{RAND}) \cap \mathrm{REC} = \mathsf{ALMOST}\text{-}\mathrm{P}_T$ [Amb86], [BG81];

(c) $\mathrm{AM} = \mathrm{NP}_T(\mathrm{RAND}) \cap \mathrm{REC} = \mathsf{ALMOST}\text{-}\mathrm{NP}_T$ [NW88];

(d) $\mathrm{PH} = \mathrm{PH}(\mathrm{RAND}) \cap \mathrm{REC} = \mathsf{ALMOST}\text{-}\mathrm{PH}$ [NW88];

(e) $\mathrm{PSPACE} = \mathrm{PQH}(\mathrm{RAND}) \cap \mathrm{REC} = \mathsf{ALMOST}\text{-}\mathrm{PQH}$, where $A \leq^{\mathrm{PQH}} B$ if and only if $A \in \mathrm{PH}(\mathrm{QBF} \oplus B)$ [BW81], [BBS85].

Proposition 3.2 shows that for appropriate $\mathcal{R}$, $\mathsf{ALMOST}\text{-}\mathcal{R}$ is precisely the "recursive part" of $\mathcal{R}(\mathrm{RAND})$. Since this work is part of the general study of complexity classes, it is particularly significant that $\mathsf{ALMOST}\text{-}\mathcal{R}$ is the recursive part of $\mathcal{R}(\mathrm{RAND})$.

By definition of $\mathsf{ALMOST}\text{-}\mathcal{R}$, a language L is in $\mathsf{ALMOST}\text{-}\mathcal{R}$ if and only if $\mathrm{Prob}[\{J \mid L \in \mathcal{R}(J)\}] = 1$. By Proposition 3.1, for any recursive L, $\mathrm{Prob}[\{J \mid L \in \mathcal{R}(J)\}] = 1$ if and only if there exists $J \in \mathrm{RAND}$ such that $L \in \mathcal{R}(J)$. Thus, $\mathsf{ALMOST}\text{-}\mathcal{R} \subseteq \mathcal{R}(\mathrm{RAND})$, and a recursive language L is in $\mathsf{ALMOST}\text{-}\mathcal{R}$ if and only if there exists $J \in \mathrm{RAND}$ such that $L \in \mathcal{R}(J)$; hence, $\mathsf{ALMOST}\text{-}\mathcal{R} = \cup_{B \in \mathrm{RAND}}(\mathcal{R}(B) \cap \mathrm{REC})$. This statement can be strengthened.

Kautz [Kau91] has developed a number of results about RAND. A modification of one of his results leads to a variation on Proposition 3.2.

LEMMA 3.3. *If* $\mathbf{X}$ *is recursively open,* $\text{Prob}[\mathbf{X}] < 1$, *and* $\mathbf{X}$ *is closed under finite translation, then* $\mathbf{X} \cap \text{RAND} = \emptyset$.

Proof. Assume that $\mathbf{X}$ is recursively open, $\text{Prob}[\mathbf{X}] < 1$, $A \subseteq \{0, 1\}^\omega$, and $A \in \mathbf{X}$. Since $\mathbf{X}$ is closed under finite translation, for all $w \in \{0, 1\}^*$, $A = \{w\} \cdot B$ implies $B \in \mathbf{X}$. It suffices to show that $A \notin \text{RAND}$.

Fix a positive integer r such that $\text{Prob}[\mathbf{X}]^r < 1/2$. Since $\mathbf{X}$ is recursively open, there is a recursively enumerable (r.e.) set C such that C is an instantaneous code (i.e., a prefix-free code) and $\mathbf{X} = C \cdot \{0, 1\}^\omega$. For each $j \in \mathbf{N}$, let $C^j = \{w_1 w_2 \cdots w_j \mid \text{each } w_i \in C\}$. Then each C^j is r.e. and there is a total recursive function g such that $W_{g(k)} = C^{rk}$ for all $k \in \mathbf{N}$.

We show that $\{A\} \subseteq W_{g(k)} \cdot \{0, 1\}^\omega$ and for each k, $\text{Prob}[W_{g(k)} \cdot \{0, 1\}^\omega] < 2^{-k}$. This implies that $\{A\}$ is a constructive null set so that $A \notin \text{RAND}$.

For the first, it suffices to prove that $A \in C^j \cdot \{0, 1\}^\omega$ for all $j \in \mathbf{N}$. This can be done by induction. It is trivial for $j = 0$. Assume that $A \in C^j \cdot \{0, 1\}^\omega$ for some j. Then $A = \{w\} \cdot B$ for some $w \in C^j$ and $B \subseteq \{0, 1\}^\omega$. By choice of A, $B \in \mathbf{X}$ and so $B = \{v\} \cdot B'$ for some $v \in C$ and $B' \subseteq \{0, 1\}^\omega$. Then $A = \{wv\} \cdot B'$ and $wv \in C^{j+1}$, so $A \in C^{j+1} \cdot \{0, 1\}^\omega$.

For the second, notice that $\text{Prob}[W_{g(k)} \cdot \{0, 1\}^\omega] = \text{Prob}[C^{rk} \cdot \{0, 1\}^\omega] = (\text{Prob}[C \cdot \{0, 1\}^\omega])^{rk} = (\text{Prob}[\mathbf{X}])^{rk} < (1/2)^k = 2^{-k}$. □

The following is a simple version of a result of Kautz [Kau91].

THEOREM 3.4. *Let* $\mathbf{X}$ *be a class that is closed under finite translation and finite variation. If* $\mathbf{X}$ *is an intersection of recursively open sets or a union of recursively closed sets, then* $\mathbf{X} \cap \text{RAND} = \emptyset$ *or* $\text{RAND} \subseteq \mathbf{X}$.

Proof. If $\mathbf{X}$ is closed under finite variation, then by the Kolmogorov 0–1 law, $\text{Prob}[\mathbf{X}] = 0$ or $\text{Prob}[\mathbf{X}] = 1$. If $\text{Prob}[\mathbf{X}] = 1$, then $\text{RAND} \subseteq \mathbf{X}$ (Corollary 3 of [BLW94]).

If $\text{Prob}[\mathbf{X}] = 0$, then $\mathbf{X} = \bigcap_{k=0}^{\infty} Y_k$ for some choice of recursively open Y_i, $i \geq 0$. Since $\text{Prob}[\mathbf{X}] = 0$, there exists $m \in \mathbf{N}$ such that the set $\mathbf{Y} = \bigcap_{k=0}^{m} Y_k$ has $\text{Prob}[\mathbf{Y}] < 1$. This set $\mathbf{Y}$ is recursively open. Let $A \in \mathbf{X}$. Then every tail of A is in $\mathbf{X} \subseteq \mathbf{Y}$ so that by Lemma 3.3, $A \notin \text{RAND}$. Hence, $\mathbf{X} \cap \text{RAND} = \emptyset$. □

Now we have the strong variation on Proposition 3.2.

THEOREM 3.5 (the random oracle characterization). *If* $\mathcal{R}$ *is an appropriate reducibility, then for every* $B \in \text{RAND}$, $\textsf{ALMOST-}\mathcal{R} = \mathcal{R}(B) \cap \text{REC}$.

Proof. Since $\textsf{ALMOST-}\mathcal{R} = \mathcal{R}(\text{RAND}) \cap \text{REC}$, it is trivial that for every $B \in \text{RAND}$, $\mathcal{R}(B) \cap \text{REC} \subseteq \textsf{ALMOST-}\mathcal{R}$.

To see that the converse is true, choose any $L \in \textsf{ALMOST-}\mathcal{R}$; then since we have $L \in \textsf{ALMOST-}\mathcal{R}$, $L \in \text{REC}$ and $\text{Prob}[\mathcal{R}^{-1}(L)] = 1$. As an appropriate reducibility, $\mathcal{R}$ is bounded so that $\mathcal{R}^{-1}(L)$ is a union of recursively closed sets (Proposition 3.1) that is closed under finite translation and finite variation. Since $\text{Prob}[\mathcal{R}^{-1}(L)] = 1$, it follows by Theorem 3.4 that $\text{RAND} \subseteq \mathcal{R}^{-1}(L)$. This implies that for every $B \in \text{RAND}$, $L \in \mathcal{R}(B)$; since $L \in \text{REC}$, this shows that $L \in \mathcal{R}(B) \cap \text{REC}$. Hence, $\textsf{ALMOST-}\mathcal{R} \subseteq \mathcal{R}(B) \cap \text{REC}$. □

COROLLARY 3.6. *Let* $\mathcal{R}$ *be an appropriate reducibility.*

(a) *Language* $A \in \textsf{ALMOST-}\mathcal{R}$ *if and only if* A *is recursive and for every* $B \in$ RAND, $\text{A} \in \mathcal{R}(B)$.

(b) *For every two languages* $C, D \in \text{RAND}$, $\textsf{ALMOST-}\mathcal{R} \subseteq \mathcal{R}(C) \cap \mathcal{R}(D)$.

Theorem 3.5 shows that for any two languages C, D in RAND, the recursive part of $\mathcal{R}(C)$ is precisely the same as the recursive part of $\mathcal{R}(D)$ since both are equal to $\textsf{ALMOST-}\mathcal{R}$. Thus, no specific language in RAND plays a special role as an oracle for any language in $\textsf{ALMOST-}\mathcal{R}$.

Theorem 3.5 generalizes the characterization of BPP due to Lutz [Lut92]. Notice that it follows immediately from the definitions (without using Theorem 3.5) that for every $B \in$ RAND, $\text{P} = \text{P}_{\text{btt}}(B) \cap \text{REC}$ and $\text{PH} = \text{PH}(B) \cap \text{REC}$, but no similar characterizations are known for classes of the form $\textsf{ALMOST-}\mathcal{R}$ without using Theorem 3.5.

4. The independent pair characterization. The theory of reducibilities has been used successfully in complexity theory to find upper bounds on the complexity of various languages. When a language A is known to be reducible to a language B, then it is sometimes the case that the inherent computational complexity of A can be bounded above by some function that bounds the inherent computational complexity of B. What can be said when language B is in RAND? It is known [Gac86] that for every language A, there is a language $B \in$ RAND such that A is Turing reducible to B. Thus, simply knowing that a language is reducible to language in RAND tells one nothing about the computational complexity of the given language. In addition, knowing that a reduction of A to B, $B \in$ RAND, is witnessed by a machine whose running time is bounded by some total recursive function still does not guarantee that A is recursive or is in some specific complexity class; this follows from the fact that if the reduction is reflexive, then B is reducible to itself within the same time bound. Recall that no recursively enumerable language (hence, no recursive language) can be in RAND.

One approach to gaining information about the complexity of the languages reducible to a language in RAND is to consider appropriate reducibilities as described in §3. Thus, consider the case where A is in ALMOST-$\mathcal{R}$ for some appropriate $\mathcal{R}$. Then A is recursive and there is a language $B \in$ RAND such that $A \in \mathcal{R}(B)$. This is enough to show that there is an upper bound on the complexity of A which depends only on the reducibility $\mathcal{R}$, that is, for every $B \in$ RAND and every $A \in \mathcal{R}(B)$, there is a machine that recognizes A and whose running time is bounded above by $f_{\mathcal{R}}$. This fact follows from a straightforward simulation based on the definition of $\mathcal{R}$ as a bounded reducibility. Hence, it is stated without proof, leaving the details as an exercise.

THEOREM 4.1. *For any appropriate reducibility $\mathcal{R}$, there is a total recursive function $f_{\mathcal{R}}$ that depends only on $\mathcal{R}$ such that every language in* ALMOST-$\mathcal{R}$ *can be recognized deterministically in time bounded by $f_{\mathcal{R}}$.*

Now we consider a special type of algorithmically random language.

Let $\mathcal{R}$ be a reducibility and $\mathbf{C}$ a class of languages. A pair (A, B) of languages is an *$\mathcal{R}$-minimal pair for* $\mathbf{C}$ if $A \notin \mathbf{C}$, $B \notin \mathbf{C}$, and for every language D, $D \in \mathcal{R}(A)$ and $D \in \mathcal{R}(B)$ imply $D \in \mathbf{C}$. (See [Amb87].)

Most of the research on minimal pairs that has been carried out in complexity theory has focused on minimal pairs for well-studied classes like P and NP with respect to reducibilities computed in polynomial time (for examples, see [Amb87], [Bre78], [Lad75], [Sch84]). However, this topic has arisen in the context of random oracles [Amb86] and [TW89]. Recently, Lutz [Lut92] developed a characterization of languages in BPP in terms of "independent pairs of algorithmically random languages," a characterization that can be put into the framework of minimal pairs and that here is generalized to every class of the form ALMOST-$\mathcal{R}$ for every appropriate $\mathcal{R}$.

Let A and B be languages such that $A \oplus B \in$ RAND. Then (A, B) is an *independent pair of algorithmically random languages.* (Here, $A \oplus B$ denotes the "join" of A and B, $A \oplus B = \{0x, 1y \mid x \in A,\ y \in B\}$.)

In the present paper we abbreviate the name as "independent pair."

Lutz observed that choosing A and B independently from Ω is equivalent to choosing the pair (A, B) from the product space $\Omega \times \Omega$ with the probability distribution given by $\text{Prob}[X \times Y] = \text{Prob}[X] \cdot \text{Prob}[Y]$ for all $X, Y \in \Omega$. Thus, almost every pair of languages is an independent pair, that is, $\text{Prob}[\{(A, B) \mid (A, B) \text{ is an independent pair}\}] = 1$.

Since $A \oplus B \in$ RAND implies that both A and B are in RAND, the name "independent pair of algorithmically random languages" is justified. Note that there exist $A, B \in$ RAND such that $A \oplus B \notin$ RAND; an example is $A \oplus \overline{A}$ where $A \in$ RAND.

Now we have the second variation on Proposition 3.2.

THEOREM 4.2 (the independent pair characterization). *Let $\mathcal{R}$ be an appropriate reducibility. For every independent pair (A, B),* $\mathsf{ALMOST}\text{-}\mathcal{R} = \mathcal{R}(A) \cap \mathcal{R}(B)$.

Proof. To show that $\mathcal{R}(A) \cap \mathcal{R}(B) \subseteq \mathsf{ALMOST}\text{-}\mathcal{R}$, it suffices to show that every independent pair is an $\mathcal{R}$-minimal pair for $\mathsf{ALMOST}\text{-}\mathcal{R}$. The proof is generalization of the proof of the main result of Lutz [Lut92]. Since Lutz was discussing only polynomial time-bounded Turing reducibilities, that proof must be altered for general appropriate reducibilities $\mathcal{R}$.

Let (A, B) be an independent pair. Let D be a language such that $D \in \mathcal{R}(A)$ and $D \in \mathcal{R}(B)$. Assume that $D \notin \mathsf{ALMOST}\text{-}\mathcal{R}$. The desired result follows by proving that $\{A \oplus B\}$ has a constructive null cover and so $A \oplus B$ is not algorithmically random.

Let M_a, M_b be machines witnessing $D \in \mathcal{R}(A)$ and $D \in \mathcal{R}(B)$. Since $\mathcal{R}$ is a bounded reducibility, one can assume that there exist total recursive functions f_a and f_b bounding the running times of M_a and M_b. Fix a strictly increasing function g such that for all x, $g(|x|) \geq \max\{f_a(|x|), f_b(|x|)\}$. Then for all x and all strings y that are query strings in either M_a's or M_b's computation on x, $|y| \leq g(|x|)$. For each natural number n, let $K(n) = 2^{g(n)} - 1$ and $N(n) = 2K(n) + 1$. In this proof certain subsets of $\{0, 1\}^{<g(n)}$ will be discussed; let $u, v \in \{0, 1\}^{K(n)}$ denote the characteristic strings of sets $U, V \subseteq \{0, 1\}^{<g(n)}$, respectively, and let $u \oplus v \in \{0, 1\}^{N(n)}$ denote the characteristic string of $U \oplus V$.

Now the proof follows that of Lutz.

Since (A, B) is an independent pair, both A and B are in RAND. By Corollary 3.6(b), this means that $\mathsf{ALMOST}\text{-}\mathcal{R} \subseteq \mathcal{R}(A) \cap \mathcal{R}(B)$. Hence, $\mathsf{ALMOST}\text{-}\mathcal{R} = \mathcal{R}(A) \cap \mathcal{R}(B)$. □

Lutz [Lut92] showed that for every independent pair (A, B), $\mathrm{BPP} = \mathrm{P}_T(A) \cap \mathrm{P}_T(B)$. That result appears to be the first characterization of this form for any class of the form $\mathsf{ALMOST}\text{-}\mathcal{R}$ for any specific appropriate $\mathcal{R}$.

From the fact that almost every pair of languages is an independent pair, it follows that for every $A \in$ RAND, (A, B) forms an independent pair for almost every language B.

Notice that for every $A \in$ RAND, there exist $B \in$ RAND such that (A, B) is *not* an independent pair and for some $\mathcal{R}$, $\mathcal{R}(A) \cap \mathcal{R}(B) \neq \mathsf{ALMOST}\text{-}\mathcal{R}$. For example, if $A \in$ RAND, then $\overline{A} \in$ RAND, but $A \oplus \overline{A} \notin$ RAND. For a reducibility like $\leq^{\mathrm{P}}_{\mathrm{T}}$, notice that $\mathrm{P}_T(A) = \mathrm{P}_T(\overline{A})$ so that $\mathrm{BPP} \neq \mathrm{P}_T(A) \cap \mathrm{P}_T(\overline{A})$.

As noted in §3, for every two languages C, D in RAND and every appropriate $\mathcal{R}$, the recursive part of $\mathcal{R}(C)$ is equal to the recursive part of $\mathcal{R}(D)$. This does not imply that $\mathcal{R}(C) = \mathcal{R}(D)$. For if $C \oplus D \in$ RAND, then $\mathcal{R}(C) \cap \mathcal{R}(D) \subseteq$ REC so that $\mathcal{R}(C) \cap \mathcal{R}(D)$ is properly included in both $\mathcal{R}(C)$ and $\mathcal{R}(D)$. But neither $\mathcal{R}(C)$ nor $\mathcal{R}(D)$ is included in REC so this implies that $\mathcal{R}(C) \neq \mathcal{R}(D)$. Thus, for every C, $\mathcal{R}(C) \neq \mathcal{R}(D)$ for almost every $D \in$ RAND.

As a more concrete application of Theorem 4.2, recall Theorem 5.2(c) of Tang and Watanabe [TW89]: for almost every pair (T_1, T_2) of tally sets, $BP \cdot \Sigma_k^P = \Sigma_k^P(T_1) \cap \Sigma_k^P(T_2)$. It has been an open question whether this result can be extended to pairs (A, B) of languages over $\{0, 1\}^*$ instead of over $\{0\}^*$. It is known [NW88] that $BP \cdot \Sigma_k^P = \mathsf{ALMOST}\text{-}\Sigma_k^P$ for every $k \geq 0$, and that almost every pair (A, B) is an independent pair.

COROLLARY 4.3. *For almost every pair (A, B) of languages,* $\mathsf{ALMOST}\text{-}\Sigma_k^{\mathrm{P}} = \Sigma_k^{\mathrm{P}}(A) \cap \Sigma_k^{\mathrm{P}}(B)$.

Lutz (personal communication) observed that for the well-studied complexity classes, this is the first result where the probabilistic result did not precede the more abstract result, that is, it was not known that $\mathrm{Prob}[\{(A, B) \mid \Sigma_k^P(A) \cap \Sigma_k^P(B) = BP \cdot \Sigma_k^P\}] = 1$ until Theorem 4.2 was established.

5. Remarks. These results give new characterizations of classes of the form $\mathsf{ALMOST}\text{-}\mathcal{R}$. Perhaps the most interesting applications (in terms of current interest) are the following:

(a) For every $B \in$ RAND, AM = NP(B) ∩ REC.

(b) For every pair (C, D) of languages such that $C \oplus D \in$ RAND, AM = NP(C) ∩ NP(D).

(c) For every $B \in$ RAND, ALMOST-PSPACE = PSPACE(B) ∩ REC.

(d) For every pair (C, D) of languages such that $C \oplus D \in$ RAND, we have ALMOST-PSPACE = PSPACE(C) ∩ PSPACE(D).

Characterizations (c) and (d) are of particular interest since no intrinsic characterization of ALMOST-PSPACE is known.

It should be noted that in Theorems 3.5 and 4.2, the reducibilities are bounded. If this condition is omitted, then things change. For example, let us write $A \in$ REC(B) if $A \leq_T B$. Then REC(RAND) ∩ REC = REC and ALMOST-REC = REC. In addition, Kautz (personal communication) has observed that a result of Kucera [Kuc86] (also see Chapter III of [Kau91]) shows that there exist languages A and B in RAND such that REC is a proper subset of REC(A) ∩ REC(B). Thus, the independent pair characterization requires that the reducibility be bounded.

Acknowledgments. I wish to thank Steven Kautz, Ker-I Ko, Jack Lutz, Elvira Mayordomo, and Greg Perkins for their helpful suggestions.

REFERENCES

[Amb86] K. Ambos-Spies, *Randomness, relativizations, and polynomial reducibilities*, in Proc. 1st Conference on Structure in Complexity Theory, Lecture Notes Comput. Sci. 223, Springer-Verlag, Berlin, New York, 1986, pp. 23–34.

[Amb87] ———, *Minimal pairs for polynomial-time reducibilities*, in Computation Theory and Logic, E. Börger, ed., Lecture Notes Comput. Sci. 270, Springer-Verlag, Berlin, New York, 1987, pp. 23–34.

[BBS85] J. Balcázar, R. V. Book, and U. Schöning, *On bounded query machines*, Theoret. Comput. Sci., 40 (1985), pp. 237–243.

[BDG88] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity* I, Springer-Verlag, Berlin, New York, 1988.

[BDG90] ———, *Structural Complexity* II, Springer-Verlag, Berlin, New York, 1990.

[BG81] C. Bennett and J. Gill, *Relative to a random oracle,* $P^A \neq NP^A \neq$ *co*-NP^A *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[BLW94] R. Book, J. Lutz, and K. Wagner, *An observation on probability versus randomness with applications to complexity classes*, Math. Systems Theory, 27 (1994), pp. 201–209.

[BW81] R. Book and C. Wrathall, *Bounded query machines: on* NP( ) *and* NPQUERY( ), Theoret. Comput. Sci., 15 (1981), pp. 41–50.

[Bre78] S. Breidbart, *Structure of complexity classes*, Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1978.

[Cai89] J.-Y. Cai, *With probability one, a random oracle separates* PSPACE*from the polynomial-time hierarchy*, J. Comput. System Sci., 38 (1989), pp. 68–85.

[Gac86] P. Gacs, *Every sequence is reducible to a random one*, Inform. and Control, 70 (1986), pp. 186–192.

[Kau91] S. Kautz, *Degrees of random sets*, Ph.D. thesis, Department of Mathematics, Cornell University, Ithaca, NY, 1991.

[Ko86] K. Ko, *On the notion of infinite pseudorandom sequences*, Theoret. Comput. Sci., 48 (1986), pp. 9–33.

[Ko92] ———, *A note on the instance complexity of pseudorandom sets*, Proc. 7th IEEE Conference on Structure in Complexity Theory, 1992, pp. 327–337.

[Kuc86] A. Kucera, *An alternative, priority-free solution to Post's problem*, in Mathematical Foundations of Computer Science, J. Gruska, B. Rovan, and P. Jorrand, eds., Lecture Notes Comput. Sci. 233, Springer-Verlag, Berlin, New York, 1986, pp. 493–500.

[Lad75] R. Ladner, *On the structure of polynomial time reducibilities*, J. Assoc. Comput. Mach., 22 (1975), pp. 155–171.

[Lut92] J. Lutz, *On independent random oracles*, Theoret. Comput. Sci., 92 (1992), pp. 301–307.

[Mar66] P. Martin-Löf, *On the definition of random sequences*, Inform. and Control, 9 (1966), pp. 602–619.

[NW88] N. Nisan and A. Wigderson, *Hardness vs. randomness*, Proc. 29th IEEE Symp. Found. of Comput. Sci., IEEE, Washington, D.C., 1988, pp. 2–11.

[Sch84] U. Schöning, *Minimal pairs for* P, Theoret. Comput. Sci., 31 (1984), pp. 41–48.

[TW89] S. Tang and O. Watanabe, *On tally relativizations of* BP-*complexity classes*, SIAM J. Comput., 18 (1989), pp. 449–462.

SIAM J. COMPUT.
Vol.23, No.6, pp. 1283–1312, December 1994

A FAST ALGORITHM FOR OPTIMUM HEIGHT-LIMITED ALPHABETIC BINARY TREES*

LAWRENCE L. LARMORE† AND TERESA M. PRZYTYCKA‡

Abstract. In this paper, an $O(nL \log n)$-time algorithm is presented for construction of an optimal alphabetic binary tree with height restricted to L. This algorithm is an alphabetic version of the Package Merge algorithm, and yields an $O(nL \log n)$-time algorithm for the alphabetic Huffman coding problem. The Alphabetic Package Merge algorithm is quite simple to describe, but appears hard to prove correct.

Garey [*SIAM J. Comput.*, 3 (1974), pp. 101–110] gives an $O(n^3 \log n)$-time algorithm for the height-limited alphabetic binary tree problem. Itai [*SIAM J. Comput.*, 5 (1976), pp. 9–18] and Wessner [*Inform. Process. Lett.*, 4 (1976), pp. 90–94] independently reduce this time to $O(n^2 L)$ for the alphabetic problem. In [*SIAM J. Comput.*, 16 (1987), pp. 1115–1123], a rather complex $O(n^{3/2} L \log^{1/2} n)$-time "hybrid" algorithm is given for length-limited Huffman coding. The Package Merge algorithm, discussed in this paper, first appeared in [*Tech. Report*, 88-01, ICS Dept. Univ. of California, Irvine, CA], but without proof of correctness.

Key words. optimal tree, weighted binary tree

AMS subject classification. 68P20

1. Introduction.

The classic *prefix-free binary coding* problem, also called the *Huffman coding* problem, was introduced, and definitively solved, by Huffman in 1952 [9]. Given a set of items ("symbols"), $A_1, \dots A_n$, where each symbol A_i has an associated "frequency," $a_i \geq 0$, find a binary string for each symbol, $code(A_i)$, minimizing the *expected code length*

$$\sum_{i=1}^{n} a_i |code(A_i)|,$$

where $|w|$ refers to the length of a string w, subject to the *prefix property*—i.e., no $code(A_i)$ is a prefix of any other $code(A_j)$. We refer to codes with the prefix property as *Huffman* codes.

The prefix property enables a bit string to be uniquely decoded. Huffman's algorithm computes a Huffman code of minimal expected length in $O(n \log n)$ time, but only linear time if the symbols are already sorted by frequency.

The "alphabetic" version of the problem introduces the one additional restriction that $code(A_i) < code(A_j)$ if $i < j$, where the relation "<" on strings refers to lexical ordering. $O(n \log n)$-time algorithms for the alphabetic problem have been given by Hu and Tucker and by Garsia and Wachs [5], [7], [8].

These algorithms, as well as subsequent algorithms for additional variations of the problem, make use of the *binary tree* representation of prefix-free codes.

The set of all bit strings (i.e., strings over {0,1}) is an infinite binary tree, where the empty string is the root, and the left and right children of any string w are $w0$ and $w1$, respectively. Any code with the prefix property can be mapped to the (finite) binary tree consisting of all $code(A_i)$ and all ancestors of those bit strings. The symbol A_i is then considered to be attached to the leaf $code(A_i)$. Note that the depth of a symbol in the tree is precisely the length of its code string.

*Received by the editors May 26, 1992; accepted for publication (in revised form) September 20, 1993.

†Department of Computer Science, University of Nevada, Las Vegas, Nevada 89154-4019 (larmore@earl.cs.unlv.edu). This research was done while the author was at the Department of Computer Science, University of California, Riverside, California 92521. This research was supported by NSF grant CCR9112067.

‡Department of Mathematics and Computer Science, Odense University, DK-5230 Odense M, Denmark (przytyck@imada.ou.dk). A part of this work was done while the author was visiting University of California, Riverside, California 92521.

Thus, the alphabetic Huffman coding problem is equivalent to the optimal binary search tree problem with the restriction that all data are in the leaves [8], [11]. In this paper, we refer to that problem as the *alphabetic binary tree* problem.

When the problem is expressed in terms of trees, we usually say *weighted depth* or *weighted path length* instead of *expected code length.*

The input for one instance of the alphabetic binary tree problem is a list of nonnegative *weights*, $w_1, \dots w_n$, and the output is a full binary tree[1] with n leaves, represented by its list of leaf depths, called its *leaf sequence*, $\ell_1, \dots \ell_n$, chosen so that the weighted depth $\sum_{i=1}^n w_i \ell_i$ is minimized.

1.1. Length-limited Huffman coding. The *length-limited Huffman coding* problem is introduced by Gilbert [6]. This is the Huffman coding problem with the restriction that every code string $code(A_i)$ must have length no greater than a given constant L. (In an application, L might be the size of a register.)

The alphabetic version of length-limited Huffman coding is equivalent to the optimal binary search tree problem with all data in the leaves, where the height of the tree is restricted. It is the latter formulation that we use in this paper, which we refer to as the *height-limited alphabetic binary tree* problem.

Previous results. Garey [4] gives an $O(n^2 \log n)$-time algorithm for the length-limited Huffman coding problem, and an $O(n^3 \log n)$-time algorithm for the height-limited alphabetic binary tree problem.

Itai [10] and Wessner [16] independently reduce this time to $O(n^2 L)$ for the alphabetic problem, essentially by showing that Garey's monotonicity lemma extends to the alphabetic case. Lemma 8.1 in this paper is a generalization of Garey's monotonicity lemma.

In [12], a rather complex $O(n^{3/2} L \log^{1/2} n)$-time "hybrid" algorithm is given for length-limited Huffman coding.

In [13] and [14], the *Package Merge* algorithm is introduced for length-limited Huffman coding. A radical departure from the dynamic programming methods of [4], [10], [12], and [16], the $O(nL)$-time Package Merge algorithm returns to the original greedy approach of Huffman's algorithm [9].

The alphabetic version of the Package Merge algorithm, an $O(nL \log n)$-time algorithm for the alphabetic Huffman coding problem, and the subject of this paper, is quite simple to describe (see §2), but appears hard to prove correct. The algorithm first appeared in [13] and [15], but without proof of correctness.

In this paper, we describe the Package Merge algorithm for a more general problem, which we call the *weighted binary tree* problem.

An instance of the weighted binary tree problem consists of a doubly indexed *weight matrix* $w_{i,\ell}$, for $i = 1, \dots n$, and $\ell = 0, \dots L$, for some given $L \geq \lceil \log_2 n \rceil$, such that

(i) $w_{i,0} = 0$,

(ii) $w_{i,\ell+1} \geq w_{i,\ell}$ [monotonicity condition],

(iii) $2w_{i,\ell} \leq w_{i,\ell-1} + w_{i,\ell+1}$ [concavity condition].

The *cost* of a binary tree $\mathcal{T}$ with respect to a weight matrix $\{w_{i,\ell}\}$ is defined to be

$$|\mathcal{T}| = \sum_{i=1}^{n} w_{i,\ell_i},$$

where $\ell_1, \dots, \ell_n$ is the leaf sequence of $\mathcal{T}$. The problem is to find the tree $\mathcal{T}$ with minimal cost, given the weight matrix. We call such a tree *optimal.*

[1] A nonempty binary tree where every nonleaf node has two children. Henceforth, in this paper, we assume that all binary trees are full.

Note that the condition that $w_{i,0} = 0$ is really no restriction. A matrix $\{w_{i,\ell}\}$ without that condition can be replaced by a matrix having that condition, by the transformation $w'_{i,\ell} = w_{i,\ell} - w_{i,0}$, preserving monotonicity and concavity. Costs of trees are changed by a constant, so optimality is preserved.

The height-limited alphabetic binary tree problem reduces to the weighted binary tree problem by letting $w_{i,\ell} = \ell w_i$ for all $\ell \leq L$.

1.2. Overview of the paper. The paper is organized as follows. In §2, we give a brief but complete description of the Package Merge algorithm with no attempt to prove correctness. In §3, we introduce a new geometric interpretation of binary trees and forests, show its relation to the traditional interpretation, and prove some of its properties. In §4, we extend this geometric interpretation to weighted binary trees and forests and show a method of constructing optimal weighted binary trees (the Package Search algorithm). In §§5 and 6, we prove properties of the technique that leads to the efficient implementation of the Package Search algorithm, the Package Merge algorithm.

A different, naturally parallelizable implementation of the Package Search algorithm will be a subject of a separate paper.

2. The Package Merge algorithm. In this section, we present a complete description of the Package Merge algorithm.

Let $w_{i,\ell}$, for $i = 1, \ldots n$, and $\ell = 0, \ldots L$, for $L \geq \lceil \log_2 n \rceil$, be an instance of the weighted binary tree problem, as defined in §1.1.

We define a *tile* to be an ordered pair of integers (i, ℓ) such that $i \in [1, n]$ and $\ell \in [0, L]$. The tile (i, ℓ) is said to have *index* i, *level* ℓ, and *width* $2^{-\ell}$. We define the *weight* of (i, ℓ) to be $w_{i,\ell} - w_{i,\ell-1}$ for $\ell > 0$, and zero if $\ell = 0$. We define the *tag value* of (i, ℓ) to be $n\ell + i$. If A is a set of tiles, define its weight, denoted $|A|$, to be the sum of the weights of its members. The width of A is defined to be the sum of the widths of its members. Define the *tag value* of A to be the maximum tag value of any of its members.[2] Finally, define the index of A to be the minimum of the indices of its members.

If $\mathcal{T}$ is any binary tree, with leaf sequence $\ell_1, \ldots \ell_n$, we define the *associated set of tiles* to be

$$\mathit{skyline}(\mathcal{T}) = \{(i, \ell) : \ell \in [0, \ell_i]\}$$

which has width $(2n - 1)$ and weight equal to $\mathit{cost}(\mathcal{T})$. (We use the word "skyline" because if that set of tiles is inverted, it consists of a set of adjacent columns of various heights and the same base, like buildings in the skyline of a city. See §3.)

The Package Merge algorithm operates by finding $\mathit{skyline}(\mathcal{T})$, the minimal weight set of tiles which has width $(2n - 1)$, subject to the condition that it is a *geometric tree*, as defined in §3. (The set of tiles generated by the Package Merge algorithm will always satisfy this condition.) $\mathcal{T}$ can then be recovered from $\mathit{skyline}(\mathcal{T})$.

We recall the Hu–Tucker algorithm [8] for constructing a minimal alphabetic binary tree. Initially, there is a list of n "square nodes," each of which has a *weight*. The algorithm consists of $(n - 1)$ steps. During each step, two nodes are combined to form a new "round" node, whose weight is equal to the sum of the weights of its children. This new "round" node is inserted in place of its left child, and the two children are deleted from the list.

In the Hu–Tucker algorithm, two nodes are said to be "tentatively connected" if there is no "square" node strictly between them in the list. During each step, the two nodes that are combined are those of minimal total weight subject to the condition that they are tentatively

[2] Tag values are used only to break ties.

connected. This construction gives rise to a binary tree, $\mathcal{T}'$. The final step of the Hu–Tucker algorithm recovers the optimal alphabetic tree, $\mathcal{T}$, from $\mathcal{T}'$. The depth ℓ_i of the ith item in $\mathcal{T}$ equals the depth of the ith "square" node in $\mathcal{T}'$. We refer the reader to [8] for details.

The Package Merge algorithm has many features in common with the Hu–Tucker algorithm. "Square" nodes of Hu–Tucker correspond to tiles in the Package Merge algorithm, and "round" nodes correspond to *packages*. A level-ℓ *package*, where ℓ is a nonnegative integer, is a certain kind of tile set that has at least two members, and whose width is $2^{-\ell}$. (A precise definition will be given in §3.2.) Packages and tiles are together called *items*.

The first step of the algorithm is to let F^L be the list of all level-L tiles, ordered by index. Then, starting with $\ell = L - 1$ and ending with $\ell = 0$, *optimal packages* of level ℓ are formed by combining the items in $F^{\ell+1}$ according to certain rules. These optimal level-ℓ packages are then merged with the list of level-ℓ tiles to form F^ℓ, a list ordered by index, where a tile is always in front of a package of the same index.

The *packaging* step, which occurs at each of the L levels, consists of as many as $(n - 1)$ steps in which two level-$(\ell + 1)$ items are selected, then combined to form a level-ℓ package. The two items are then deleted from $F^{\ell+1}$. The rule is to select that pair of smallest total weight, subject to the condition that the pair is "tentatively connected." A pair of items in $F^{\ell+1}$ is *tentatively connected* if there is no tile which is strictly between them in the list. Since items are deleted as they are selected, there will always be at least one tentatively connected pair if the list has at least two items.

Note that there can be at most one item left in $F^{\ell+1}$ at the end of the packaging step. This item, which we call an "orphan," is simply discarded.

The set *skyline*$(\mathcal{T})$ will be simply the union of the items in F^0, from which $\mathcal{T}$ can be easily recovered.

Ties are broken using the tag values. In particular, if two items have equal weight, that with the higher tag value is considered heavier.

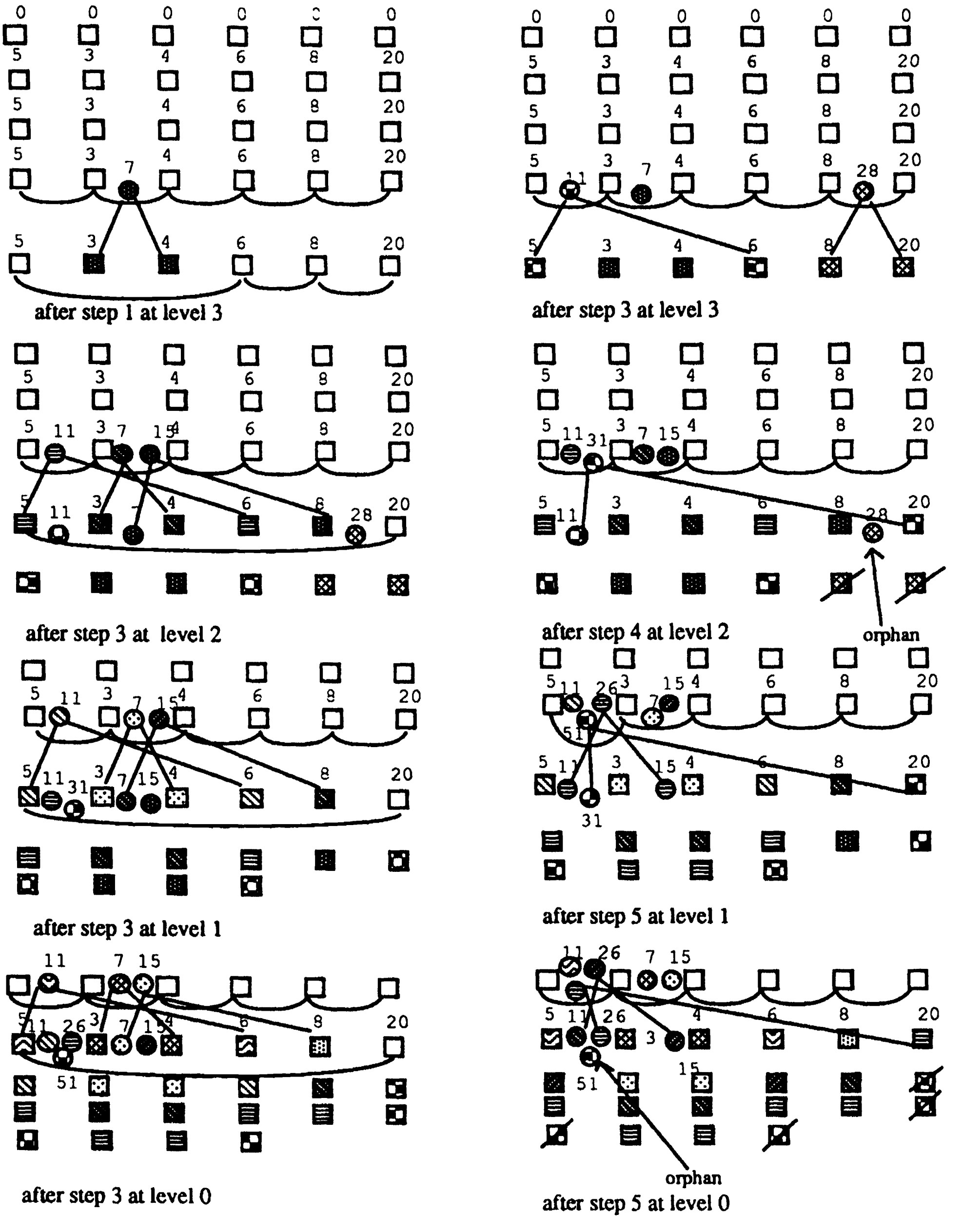
The Package Merge algorithm is illustrated in Figs. 1 and 2. We give a pseudo-code below.

```
ALGORITHM Package Merge (L)
1. For all $\ell \in [1, L]$, $F^\ell \leftarrow$ all level-$\ell$ tiles
2. $F^0 \leftarrow$ empty list
3. for $\ell = L - 1$ downto 0 do
      3.1. while $|F^{\ell+1}| \geq 2$ do
               3.2.1. $(p_1, p_2) \leftarrow$ least weight tentatively connected pair in $F^{\ell+1}$
               3.2.2. $p \leftarrow p_1 \cup p_2$
               3.2.3. insert $p$ into $F^\ell$
               3.3.4. delete $p_1, p_2$ from $F^{\ell+1}$
4. skyline$(\mathcal{T}) \leftarrow$ union of all members of $F^0$
5. Recover $\mathcal{T}$ from skyline$(\mathcal{T})$
```

The question of whether the Package Merge algorithm is correct if ties are broken arbitrarily is open.

We suggest using a list of mergeable priority queues to represent $F^{\ell+1}$. A pair of items will be tentatively connected if and only if they are both in the same priority queue. Using this data structure, it takes $O(\log n)$ time to select the minimal weight, tentatively connected pair. Since that selection must occur at most $(n - 1)$ times on each of L levels, the time complexity of the Package Merge algorithm is $O(nL \log n)$.

3. Geometric trees. In this section, we introduce a new geometric interpretation of binary trees and forests, and show its relation to the traditional interpretation.


after step 1 at level 3

after step 3 at level 3

after step 3 at level 2

after step 4 at level 2

after step 3 at level 1

after step 5 at level 1

after step 3 at level 0

after step 5 at level 0

FIG. 1. *Constructing skyline*$(\mathcal{T})$.

Consider an $n \times (L + 1)$ rectangle in the plane, divided into unit squares which we call *tiles*. (See Fig. 3.) A "geometric tree" or "geometric forest" will be defined to be a certain kind of connected union of tiles. We refer to the tile in the ℓth row and the ith column as $s_{i,\ell}$, and we say that $s_{i,\ell}$ has *index* i and *level* ℓ. We adopt the conventions that levels range from 0 to L, from the top down, while indices range from 1 to n, from left to right.

Two tiles are *adjacent* if they share a boundary edge, not just a corner. Thus, $s_{i,\ell}$ is adjacent to $s_{i+1,\ell}$ and to $s_{i,\ell+1}$, but not to $s_{i+1,\ell+1}$. A *path of tiles* is a sequence of tiles in

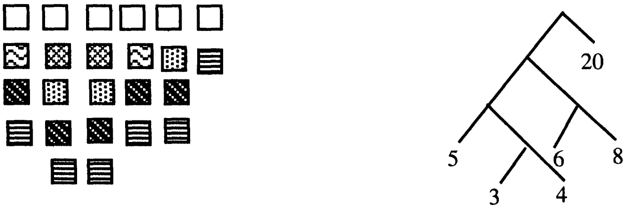


FIG. 2. *Recovering the tree $\mathcal{T}$ from skyline($\mathcal{T}$).*

which consecutive tiles are adjacent. We say that a set of tiles A is *connected* if for any two tiles $a_1, a_2 \in A$ there exists a path of tiles between a_1 and a_2 that is entirely contained in A.

For any $M \leq N$ and any set of tiles S, we define $S_{[M,N]}$ to be the set of all members of S whose indices are in the interval $[M, N]$. Formally,

$$S_{[M,N]} = \{s_{i,\ell} \in S \mid M \leq i \leq N\}.$$

DEFINITION. *We define a skyline at level ℓ (or level-ℓ skyline, for short) to be a connected set of tiles A such that*

1. *A contains no tile at any level $\ell' < \ell$, and*
2. *if $s_{i,\ell'+1} \in A$ for any $\ell' \geq \ell$, then $s_{i,\ell'} \in A$.*

A skyline looks like an upside-down bar graph. It consists of one or more columns of tiles which are adjacent in the horizontal dimension, such that the top tile of each column is at the same level (see Fig. 3).

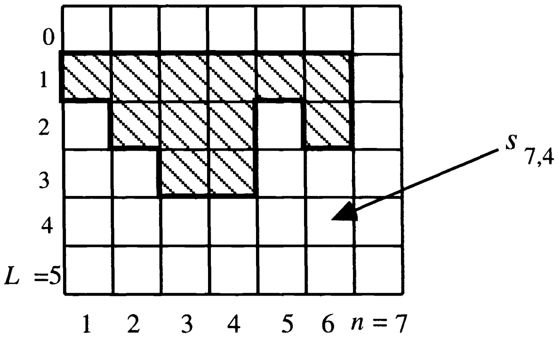


FIG. 3. *A skyline of level* 1.

For any ℓ, we define B^ℓ to be the set of all level-ℓ tiles. The *base* of a level-ℓ skyline A is defined to be the set of all its level-ℓ tiles, i.e., $A \cap B^\ell$. By connectivity, this base will always consist of a contiguous partial row of tiles $s_{M,\ell}, \ldots s_{N,\ell}$. We call $[M, N]$ the *base interval* of A, and we say that A has *base length* $(N - M + 1)$. We also say that A is a skyline *over* $[M, N]$.

Certain skylines will be called *geometric trees*, and certain ones will be called *geometric forests*. These concepts are invariant under translation up, down, or sideways, but not under rotation.

We give a recursive definition of a *geometric forest*, and a *geometric tree* is simply a geometric 1-forest.

DEFINITION. (i) A level-ℓ tile is a level-ℓ geometric tree.

(ii) A level-ℓ skyline which is the exact union of k level-ℓ geometric trees is a level-ℓ geometric k-forest.

(iii) If A is a level-$(\ell + 1)$ geometric 2-forest over $[M, N]$, then $A \cup B^{\ell}_{[M,N]}$ is a level-ℓ geometric tree.

Note that a geometric k-forest A is simply the disjoint union of k geometric trees that have adjacent bases.

The following lemma follows immediately from the above definition.

LEMMA 3.1. *If A is a level-$(\ell + 1)$ skyline over $[M, N]$, then $A \cup B^{\ell}_{[M,N]}$ is a level-ℓ geometric k-forest if and only if A is a level-$(\ell + 1)$ geometric $2k$-forest.*

Usage of the words "tree" and "forest" is justified by the following correspondence lemma.

LEMMA 3.2. *There is a one-to-one correspondence between level-ℓ geometric k-forests over $[M, N]$ and ordered k-forests of size*[3] *$(N - M + 1)$ and height at most $(L - \ell)$, as follows: If F is a geometric k-forest over $[M, N]$, then F corresponds to the forest $\mathcal{F}$ whose leaf sequence is $\ell_M, \ell_{M+1,\dots\ell_N}$, where $\ell_i + \ell$ is the greatest level of any tile in F of index i.*

Although somewhat complex in its statement, Lemma 3.2 follows immediately from the inductive definition of geometric forest. We omit the proof.

If $\mathcal{F}$ is an ordered k-forest of height h and size $(N - M + 1)$, we define the *level-ℓ representation of $\mathcal{F}$ over $[M, N]$* to be the geometric k-forest at level ℓ over $[M, N]$ that corresponds to $\mathcal{F}$ under Lemma 3.2, which exists and is unique provided $(\ell + h) \leq L$.

Figure 4 shows a binary tree, a forest, and their corresponding geometric tree and forest.

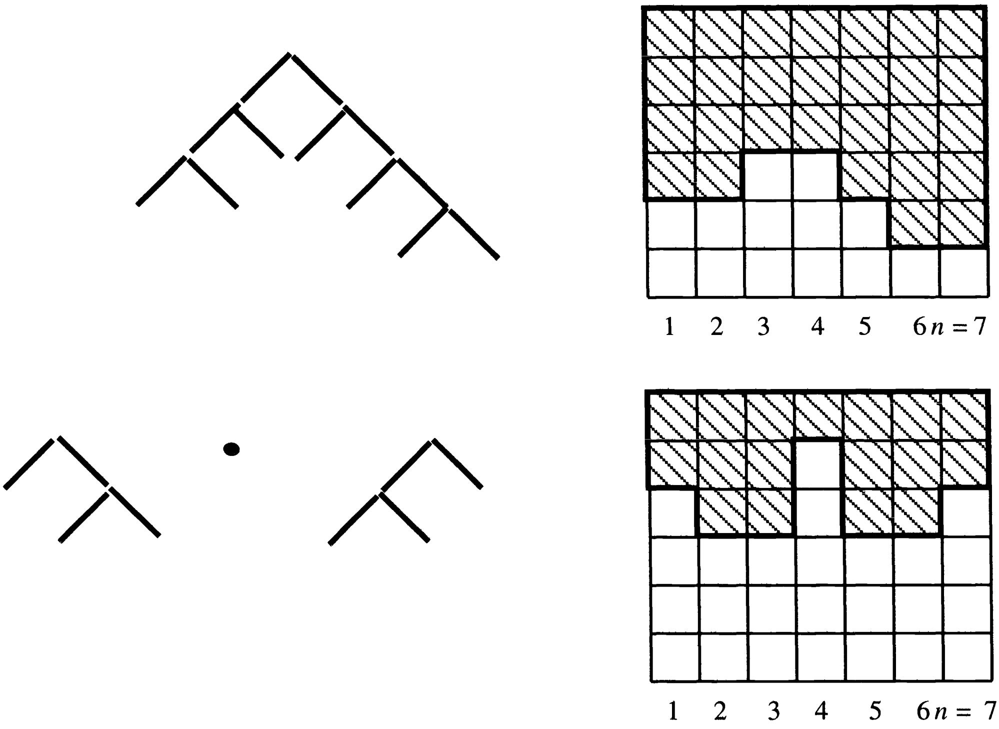


FIG. 4. *A binary tree, a forest, and their corresponding geometric tree and forest.*

[3] The *size* of a tree or forest is defined to be the number of leaves.

Henceforth, we say "tree" and "forest" instead of "geometric tree" and "geometric forest," provided the meaning is clear from context.

The recursive definitions of geometric "tree" and "forest" above allow us to build up geometric trees and forests out of smaller ones in ways which are analogous to the usual building of binary trees and binary forests out of smaller ones. But we can also build them by a radically different method—which is the essence of this paper—by changing a geometric k-forest into a geometric $(k-1)$-forest by adding an appropriate set of tiles, which we will call a *package*.

Before plunging into formalism, we ask the reader to examine Fig. 5, which shows a geometric 3-forest of size 7, which has been changed into a geometric 2-forest, also of size 7, by the addition of some tiles. It is important to note that this mutation is quite different from the usual recursive construction—the leaves of the middle tree have been divided among the two trees of the new forest.

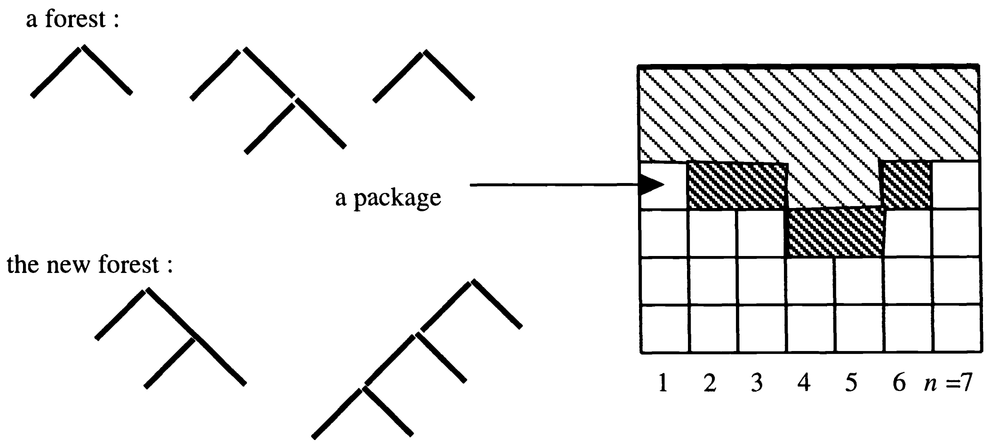


FIG. 5. *Changing a* 3*-forest into a* 2*-forest by adding a* package.

3.1. Boundaries, spanning, gaps, and cuts. If A is a nonempty set of tiles, we define *lboundary* (A) to be the smallest index of any tile which is a member of A, and we define *rboundary* (A) to be the largest index of any tile which is a member of A.

We also say that A *spans* an index r if $r \in [\mathit{lboundary}\ (A), \mathit{rboundary}\ (A)]$.

Let $\ell' \geq \ell$. We say that a set of tiles A is *level-ℓ' connected* with respect to a level-ℓ skyline Q over $[M, N]$ if

1. $A \cap Q = \emptyset$.
2. $A \cup Q$ is a level-ℓ skyline over $[M, N]$.
3. For any two tiles $a_1, a_2 \in A$ there exists a path of tiles between a_1 and a_2 that is entirely contained in $A \cup Q$, and which uses no tiles at level ℓ' or above.[4]

Remark 3.3. *If a set of tiles A is level-ℓ' connected with respect to two level-ℓ skylines, Q and Q', where $\ell' \geq \ell$, then A is level-ℓ' connected with respect to $Q \cap Q'$.*

Thus, if A is level-ℓ' connected with respect to some level-ℓ skyline Q, and if A spans r, then $s_{r,\ell'+1} \in A \cup Q$.

If Q is a level-ℓ skyline, we say that r is a *gap* of Q if $r \in [\mathit{lboundary}\ (Q), \mathit{rboundary}\ (Q)]$ and $s_{r,\ell+1} \notin Q$. If r is one of the boundary values, we say that the gap is *external*, otherwise *internal* (see Fig. 6).

[4] To be consistent with our figures, levels which are less are called "higher."

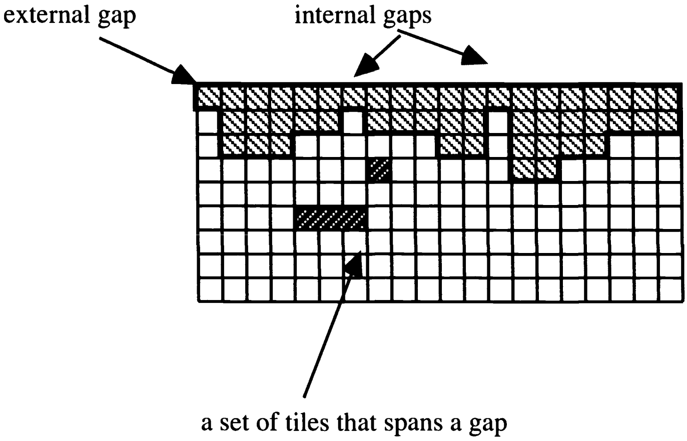


FIG. 6. *Internal and external gaps and a set of tiles that spans a gap.*

Let F be a k-forest over $[M, N]$. We define the *cuts* of F to be the right boundaries of the trees which form F. Formally, if F is the disjoint union of geometric trees $T_1, \ldots T_k$ (from left to right) at the same level as F, define $c_m(F) = \mathit{rboundary}\ (T_m)$. We also let $c_0(F) = M - 1$. We say that $(M - 1)$ and $c_k(F) = N$ are the *external* cuts of F, and that the others are *internal* cuts.

$F_{[c_s(F)+1, c_t(F)]}$ is a $(t - s)$-forest (which we call a *subforest* of F) at the same level as F, for $0 \leq s \leq t \leq k$.

LEMMA 3.4 (gap implies two cuts). *A geometric forest F has a gap at r if and only if it has cuts at $(r - 1)$ and at r.*

Proof. By the recursive definition, no tree of base length more than 1 can have a gap. If r is a gap of F, then $s_{r,\ell}$ must be one of the constituent trees of F, hence there must be a cut on either side of it. The converse follows from the fact that the only trees of base length 1 are singleton tiles. □

3.2. Packages, glue packs, and parse trees. We now give jointly recursive (but still fairly simple) definitions of two concepts: *level-ℓ item over* $[M, N]$ and *level-ℓ package over* $[M, N]$.

DEFINITION. A *level-ℓ item over* $[M, N]$ is either a level-ℓ package over $[M, N]$ or a tile $s_{i,\ell}$ where $M \leq i \leq N$. A *level-ℓ package over* $[M, N]$ is the disjoint union of two level-$(\ell+1)$ items over $[M, N]$.

Any item p can be represented with the help of a binary tree $T(p)$, called the *parse tree* of the item. The nodes of $T(p)$ are subsets of p which are themselves items at various levels. The root of $T(p)$ is p. If p is a tile, $T(p)$ has just one node. If p is a level-ℓ package consisting of the disjoint union of level-$(\ell + 1)$ items p_1, p_2, then $T(p_1)$ and $T(p_2)$ are the left and right subtrees. Note that the construction of $T(p)$ is not unique, as there may be more than one way to represent a package as the union of two items.

We warn the reader that, in spite of the similarity of the recursive definitions, items are radically different from geometric trees. In particular, they need not be connected. There is a relationship between the two concepts, though. We shall see that every level-ℓ geometric k-forest of base length d is the disjoint union of precisely d level-ℓ tiles and $(d - k)$ level-ℓ packages.

The essence of the Package Merge algorithm is to build "optimal" packages at each level ℓ by combining level-$(\ell + 1)$ tiles and optimal level-$(\ell + 1)$ packages. The optimal level-0 tree over $[1, n]$ will then be B^0, together with the disjoint union of precisely $(n - 1)$ optimal level-0 packages. The tree corresponding to this optimal geometric tree is the solution to the problem, i.e., the minimal cost tree.

DEFINITION. Let Q be a level-ℓ skyline over $[M, N]$ and p be a level-ℓ' item, $\ell' \geq \ell$, over $[M, N]$, disjoint from Q. Then p is called a *level-ℓ' glue-pack* for Q if and only if p is level-ℓ' connected with respect to Q and one of the following two conditions holds:

(i) p is a tile;

(ii) $p = p_1 \cup p_2$, where p_1 is a level-$(\ell' + 1)$ glue-pack with respect to Q, and p_2 is a level-$(\ell' + 1)$ glue-pack with respect to $Q \cup p_1$.

Figure 7 shows an example of a package that is a glue-pack with respect to a polygon and an example of a package that is not a glue-pack with respect to that polygon.

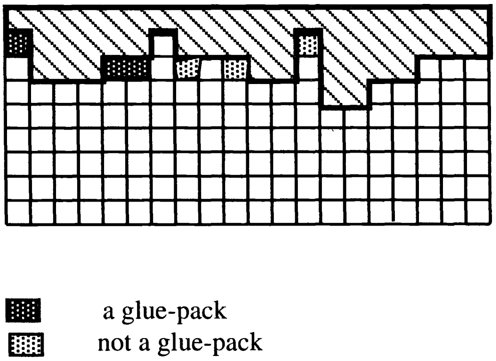


FIG. 7. *Examples of a package that is a is a glue-pack and another that is not a glue-pack with respect to a skyline.*

If p is a glue-pack with respect to some polygon Q then a parse tree for p implied by the definition of a glue-pack is called a *normal parse tree with respect to Q*. Formally, if p is a package, and if p_1, p_2 are as given in the definition, then the left and right subtrees of the normal parse tree of p must be a normal parse tree for p_1 with respect to Q, and a normal parse tree for p_2 with respect to $Q \cup p_1$, respectively. (In spite of this restriction, a normal parse tree need not be unique.)

Immediately from the definition of a glue-pack we have the following lemmas, which can be proved by induction on the parse tree of a glue-pack, using Remark 3.3.

LEMMA 3.5. *If $A \subseteq B$ are skylines, and if p is a glue-pack with respect to A and is disjoint from B, then p is a glue-pack with respect to B.*

LEMMA 3.6. *If A and B are skylines, and if p is a glue-pack with respect to A and also with respect to B, then p is a glue-pack with respect to $A \cap B$.*

LEMMA 3.7. *If p is any package such that $p_{[M,N]} = p$, then p is a glue-pack over a skyline A if and only if p is a glue-pack over $A_{[M,N]}$.*

LEMMA 3.8. *If $\ell' > \ell$, p is a level-ℓ' glue-pack with respect to a level-ℓ skyline Q over $[M, N]$, and if either $\ell' > (\ell + 1)$ or p is not a tile, then p is also a level-ℓ' glue-pack with respect to $\left(Q - B^{\ell}\right) \cup B_{[M,N]}^{(\ell+1)}$.*

Informally, the lemma below states that if p is a glue-pack that spans a gap, then it must "fill" the gap, and cannot extend beyond both boundaries of the gap.

LEMMA 3.9 (gap-filling property). *If Q is a skyline that does not contain the tile $s_{i,\ell+1}$, and if p is a level-ℓ glue-pack with respect to Q, then we have what follows.*

1. *If p spans i, then either $p = s_{i,\ell}$ or $s_{i,\ell+1} \in p$.*
2. *Either rboundary $(p) \leq i$ or lboundary $(p) \geq i$.*

Proof. The proof is by induction on the number of tiles in p. If p is a tile, the result is trivial, so assume p is a package. Let p_1, p_2 be the children of p in its parse tree. If p does not span i, the result is trivial, so assume p spans i. Since $Q \cup p$ is a skyline, $s_{i,\ell+1} \in p$, since otherwise $Q \cup p$ would not be level-ℓ connected. We can let $p_1 = s_{i,\ell+1}$. Since p_2 is a glue-pack with respect to $Q \cup p_1$ which does not contain $s_{i,\ell+2}$, the inductive hypothesis guarantees that either *rboundary* $(p_2) \leq i$ or *lboundary* $(p_2) \geq i$, and we are done. □

We now present the main result of this section.

THEOREM 3.10. *If Q is a level-ℓ k-forest over $[M, N]$ where $k > 1$ and p is a level-ℓ glue-pack with respect to Q, then $Q \cup p$ is a level-ℓ $(k - 1)$-forest over $[M, N]$.*

Proof. We prove the theorem by backwards induction on the level, ℓ, and within each level by induction on the base length.

If $\ell = L$ then $B^{\ell}_{[M,N]}$ is the only level-ℓ forest over $[M, N]$. Similarly, for any ℓ, if $M = N$ then $s_{M,\ell}$ is the only level-ℓ forest over $[M, N]$. Thus, if $\ell = L$ or the base length is 1, the theorem holds vacuously.

The method we use for the inductive step is to break Q into three subforests (a subforest can be empty) by using two cuts. One subforest F' (say, an s-forest) spans p, and the other two (say, an a-forest F'' and a b-forest F''', where $a + s + b = k$) are to the left and right of F' and are unaffected by p. We show that $F' \cup p$ is an $(s - 1)$-forest over the same base interval as F', and thus $F \cup p = F'' \cup (F' \cup p) \cup F'''$ is a $(k - 1)$-forest.

Assume that the statement of the theorem is true for all $\ell' > \ell$ and, within ℓ, for all M', N' such that $(N' - M') < (N - M)$. Let Q be a level-ℓ k-forest over $[M, N]$. Let p be a level-ℓ glue-pack with respect to Q. Note that p cannot be a tile since it must be disjoint from Q. Let $T(p)$ be a normal parse tree for p, and let p_1, p_2 be the left and right children of p in this parse tree.

Consider the following three cases, illustrated in Fig. 8.

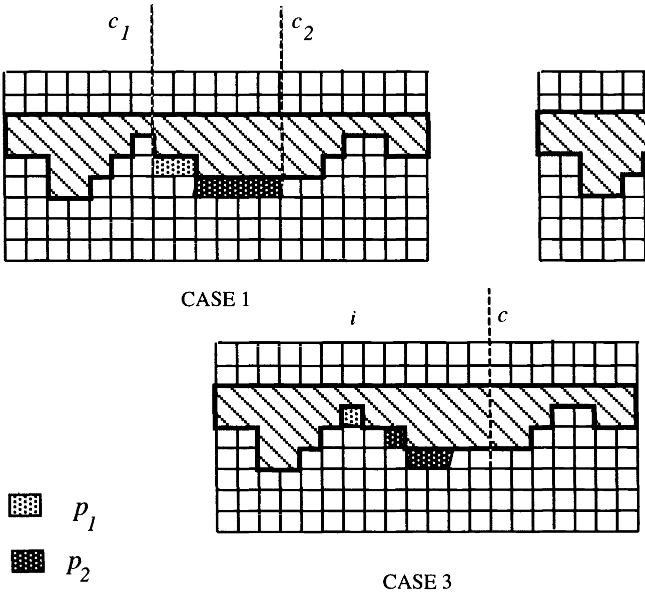


FIG. 8. *The three cases considered in the proof of Theorem* 3.10.

Case 1. Both p_1 and p_2 are level-$(\ell + 1)$ packages. Let c_1 be the largest cut that is less than *lboundary* $(p_1 \cup p_2)$ and let c_2 be the smallest cut that is greater than or equal to

$rboundary(p_1 \cup p_2)$. Let $F' = Q_{[c_1+1,c_2]}$. Since $p_1 \cup p_2$ is level-ℓ connected with respect to Q, we have, by Lemma 3.9, that Q has no gaps between c_1 and c_2. Thus, by Lemma 3.1, $F' - B^\ell$ is a level-$(\ell + 1)$ $2s$-forest. By the definition of a glue-pack, p_1 is a level-$(\ell + 1)$ glue-pack with respect to $F' - B^\ell$ and p_2 is a level-$(\ell + 1)$ glue-pack with respect to $(F' - B^\ell) \cup p_1$.

By the inductive hypothesis, $(F' - B^\ell) \cup p_1$ is a level-$(\ell + 1)$ $(2s - 1)$-forest over $[c_1 + 1, c_2]$, and $(F - B^\ell) \cup p_1 \cup p_2 = (F - B^\ell) \cup p$ is a level-$(\ell + 1)$ $(2s - 2)$-forest over the same base interval. By Lemma 3.1, $F' \cup p_1 \cup p_2 = F' \cup p$ is an $(s - 1)$-forest over $[c_1 + 1, c_2]$, and therefore $Q \cup p$ is a level-ℓ $(k - 1)$-forest over $[M, N]$.

Case 2. Both p_1 and p_2 are level-$(\ell+1)$ tiles. Let $p_1 = s_{i,\ell+1}$ and $p_2 = s_{j,\ell+1}$, and $i < j$. Then $(i - 1)$, i, $(j - 1)$, and j are cuts of Q, by Lemma 3.4. Let $F' = Q_{[i,j]}$. Then $Q_{[i+1,j-1]}$ is an $(s - 2)$-forest. By Lemma 3.1, $Q_{[i+1,j-1]} - B^\ell$ is a level-$(\ell + 1)$ $(2s - 4)$-forest over $[i + 1, j - 1]$, and thus $(Q_{[i+1,j-1]} - B^\ell) \cup p_1 \cup p_2 = (F' - B^\ell) \cup p_1 \cup p_2 = (F' - B^\ell) \cup p$ is a level-$(\ell + 1)$ $(2s - 2)$-forest over $[i, j]$.

By Lemma 3.1, $F' \cup p$ is a level-ℓ $(s - 1)$-forest over $[i, j]$. Thus $Q \cup p$ is a level-ℓ $(k - 1)$-forest over $[M, N]$.

Case 3. One of the p_i is a level-$(\ell + 1)$ tile and the other is a level-$(\ell + 1)$ package. We can assume that p_1 is a tile (let $p_1 = s_{i,\ell+1}$) and that p_2 is a package.

Then i is a gap of Q and $(i - 1)$ and i are cuts, by Lemma 3.4. By Lemma 3.9, either $rboundary(p_2) \leq i$ or $lboundary(p_2) \geq i$. Without loss of generality, the second case holds. Let c be the smallest cut of Q that is greater than or equal to *rboundary* (p_2), and let $F' = Q_{[i,c]}$. Then $Q_{[i+1,c]}$ is a level-ℓ $(s - 1)$-forest, hence, by Lemma 3.1, $Q_{[i+1,c]} - B^\ell$ is a level-$(\ell + 1)$ $(2s - 2)$-forest over $[i + 1, c]$, and $\left(Q_{[i+1,c]} - B^\ell\right) \cup s_{i,\ell+1}$ is a level-$(\ell + 1)$ $(2s - 1)$-forest over $[i, c]$. From Lemma 3.8, p_2 is a glue-pack with respect to $\left(Q_{[i+1,j]} - B^\ell\right) \cup s_{i,\ell+1}$. By the inductive hypothesis, $\left(Q_{[i+1,c]} - B^\ell\right) \cup s_{i,\ell+1} \cup p_2 = \left(Q_{[i+1,c]} - B^\ell\right) \cup p$ is a level-$(\ell + 1)$ $(2s - 2)$-forest over $[i, c]$. Thus, by Lemma 3.1, $F' \cup p$ is a level-ℓ $(s - 1)$-forest over $[i, c]$ and $Q \cup p$ is a level-ℓ $(k - 1)$-forest over $[M, N]$.

This concludes the proof of Theorem 3.10. □

4. Geometric weighted trees.

In this section we give a polygonal interpretation of weighted binary forests.

We define an instance of the *geometric tree* problem to consist of integers $n \geq 1$ and $L \geq \log_2 n$, and an assigned *weight* to every tile, as defined in §3. (For each $i \in [1, n]$ and each $\ell \in [0, L]$, we write $|s_{i,\ell}|$ to be the weight of $s_{i,\ell}$.) The weight function must satisfy

(i) $|s_{i,\ell}| \geq 0$ [nonnegativity], and

(ii) $|s_{i,\ell+1}| \geq |s_{i,\ell}|$ [monotonicity].

The problem is to find a level-0 geometric tree over $[1, n]$ which has minimum total weight.

To simplify our presentation we assume that the following hypothesis holds:

(iii) If $A \neq B$ are sets of tiles, then $|A| \neq |B|$ [uniqueness hypothesis].

We will show in §7 that the general case reduces to the case where the uniqueness hypothesis holds. Specifically, in case two sets of tiles have the same weight, we break the tie in a consistent way by imagining that each tile has a certain "infinitesimal" additional weight.

4.1. Reduction.

The weighted binary tree problem reduces to the geometric tree problem, as follows. Let $|s_{i,\ell}|$ be defined to be $w_{i,\ell} - w_{i,\ell-1}$ for $\ell > 0$, and let $|s_{i,0}| = 0$.

Nonnegativity and monotonicity follow from monotonicity and concavity, respectively, for the weighted binary tree problem.

The fact that this gives a true reduction comes from the following lemma, which follows immediately from the definition of the level-0 representation of a tree.

LEMMA 4.1. *The cost of a binary tree is equal to the weight of its level*-0 *representation over* $[1, n]$.

4.2. Optimal forests. Henceforth in this section, we assume that we are given an instance of the geometric tree problem which satisfies the uniqueness hypothesis.

Define the *optimal level-ℓ k-forest over* $[M, N]$ (denoted $F[M, N, k, \ell]$) to be that unique level-ℓ k-forest over $[M, N]$ of minimal weight, defined if and only if

$$(N - M + 1)2^{\ell-L} \leq k \leq (N - M + 1).$$

LEMMA 4.2. *Any subforest of an optimal forest is optimal.*

Proof. If $F[M, N, k, \ell]$ contains a nonoptimal subforest over a subinterval $[M', N']$, simply replace that subforest by the appropriate optimal forest and obtain an improvement of $F[M, N, k, \ell]$, which is a contradiction. □

We will also need the following lemma, whose proof is fairly complex, but related to the various "monotonicity" lemmas of [1], [4], [10]–[12], [16], and [17].

LEMMA 4.3 (interleaving property). *If M, N, k, ℓ are given where $1 \leq M < N \leq n$, $0 \leq \ell \leq L$, $(N - M + 1)2^{\ell-L} \leq k \leq N - M$, and $0 \leq m \leq k$, then*

$$c_m(F[M, N, k + 1, \ell]) \leq c_m(F[M, N, k, \ell]) \leq c_{m+1}(F[M, N, k + 1, \ell]).$$

The proof of Lemma 4.3 is given in §8.

The interleaving property has a number of immediate corollaries which allow us to relate the shapes of $F[M, N, k + 1, \ell]$ and $F[M, N, k, \ell]$.

LEMMA 4.4. *If $F[M, N, k, \ell]$ has a gap at r, then $F[M, N, k + 1, \ell]$ has a gap at r.*

Proof. By Lemma 3.4, we can write

$$\begin{aligned} c_{m-1}(F[M, N, k, \ell]) &= r - 1, \\ c_m(F[M, N, k, \ell]) &= r. \end{aligned}$$

By Lemma 4.3, $c_m(F[M, N, k + 1, \ell])$ is equal to either $(r - 1)$ or r. Those two cases are seen to be equivalent by using the natural left–right symmetry of the entire problem. Thus, without loss of generality, we may assume that $c_m(F[M, N, k + 1, \ell]) = r$. By Lemma 4.2, $F[M, N, k + 1, \ell]_{[M,r]} = F[M, r, m, \ell] = F[M, N, k, \ell]_{[M,r]}$. By Lemma 3.4, we are done. □

LEMMA 4.5. *If $F[M, N, k + 1, \ell]$ has an internal gap at r, $F[M, N, k, \ell]$ and $F[M, N, k + 1, \ell]$ have a common internal cut, either at $(r - 1)$ or at r.*

Proof. By Lemma 3.4, we can write

$$\begin{aligned} c_{m-1}(F[M, N, k + 1, \ell]) &= r - 1, \\ c_m(F[M, N, k + 1, \ell]) &= r. \end{aligned}$$

By Lemma 4.3, $c_{m-1}(F[M, N, k, \ell])$ is equal to either $(r - 1)$ or r. □

LEMMA 4.6. *If $F[M, N, k, \ell]$ and $F[M, N, k + 1, \ell]$ have a common cut, c, then either*

$$F[M, N, k, \ell]_{[M,c]} = F[M, N, k + 1, \ell]_{[M,c]}$$

or

$$F[M, N, k, \ell]_{[c+1,N]} = F[M, N, k + 1, \ell]_{[c+1,N]}.$$

Proof. Write $c = c_m(F[M, N, k, \ell])$. If $c = c_m(F[M, N, k + 1, \ell])$, then, by Lemma 4.2,

$$F[M, N, k, \ell]_{[M,c]} = F[M, N, k + 1, \ell]_{[M,c]}.$$

If $c = c_{m+1}(F[M, N, k+1, \ell])$, then, by Lemma 4.2,

$$F[M, N, k, \ell]_{[c+1,N]} = F[M, N, k+1, \ell]_{[c+1,N]}.$$

By Lemma 4.3, there are no other possibilities. □

THEOREM 4.7 (inclusion property). *For any M, N, k, ℓ such that $1 \leq M < N \leq n$, $0 \leq \ell \leq L$, and $(N - M + 1)2^{\ell - L} \leq k \leq N - M$,*

$$F[M, N, k, \ell] = F[M, N, k+1, \ell] \cup p$$

where p is the smallest weight level-ℓ package which is a glue-pack with respect to $F[M, N, k, \ell]$.

We will prove the inclusion property by backwards induction on ℓ. Thus we need to examine the dependency between optimal level-$(\ell + 1)$ forests and optimal level-ℓ forests. Such a dependency is given in Lemmas 4.8, 4.10, and 4.11 and illustrated in Fig. 9 (a), (b), and (c).

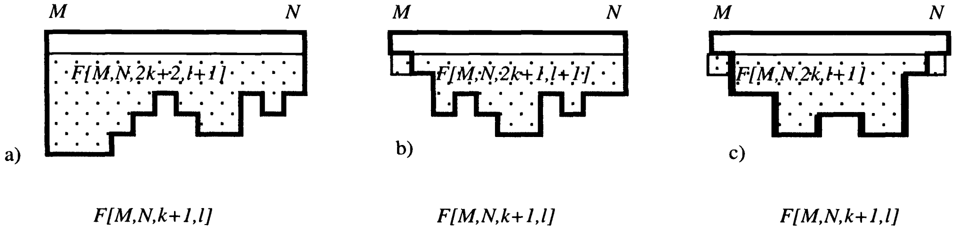


FIG. 9. *Illustration of Lemmas* 4.8, 4.10, *and* 4.11.

LEMMA 4.8. *If $F[M, N, k+1, \ell]$ does not have gaps, then $F[M, N, 2k+2, \ell+1] = F[M, N, k+1, \ell] - B^\ell$.*

Proof. If not, by Lemma 3.1, $F[M, N, 2k+2, \ell+1] \cup B^\ell_{[M,N]}$ would be a level-ℓ $(k+1)$-forest over $[M, N]$ of smaller weight than $F[M, N, k+1, \ell]$, a contradiction. □

For the proofs of Lemmas 4.10 and 4.11, we will need the following technical lemma.

LEMMA 4.9. *If F is any level-ℓ k-forest over $[M, N]$ which does not have a gap at r, where $r = M$ or $r = N$, then there exists a level-ℓ $(k+1)$-forest $F' \subseteq F$ over $[M, N]$ such that $|F'| < |F - s_{r,\ell}|$.*

Proof. Without loss of generality, $r = M$. Consider the binary forest $\mathcal{F}$ represented by the level-ℓ forest F (see Fig. 10 (a)). The first leaf, v_M, of $\mathcal{F}$ has depth $d \geq 1$. Let v be the sibling of v_M. Remove v_M from $\mathcal{F}$ together with its parent and promote the sibling of v_M (and all its descendants) one level up (see Fig. 10 (b)). The resulting binary forest has k trees. Now we add the single-element tree containing just the leaf v_M at the front of this forest. The new binary forest, which we refer to as $\mathcal{F}'$, has $(k+1)$ trees. Let F' be the level-ℓ representation of $\mathcal{F}'$ over $[M, N]$ (see Fig. 10 (c)). We consider how F' differs from F.

Removing the leaf v_M from $\mathcal{F}$ corresponds to deleting the column of all tiles of index M from the geometric representation, F. Promoting all descendants of v corresponds to deleting the bottom (i.e., greatest level) tile of F of each index in the interval $[M+1, M+t]$, where t is the number of leaves in the subtree rooted at v.

Reinserting v_M as a tree of size 1 corresponds to reinserting the tile $s_{M,\ell}$. F' can thus be obtained from F by removing a set of tiles, that contains, in particular, $s_{M,\ell+1}$. Since $|s_{M,\ell}| \leq |s_{M,\ell+1}|$, we are done. □

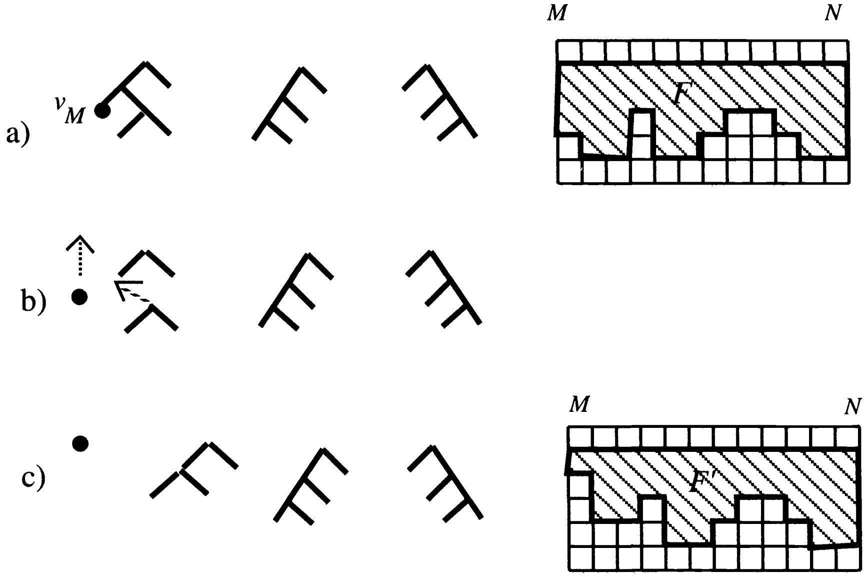


FIG. 10. *Illustration of the proof of Lemma* 4.9.

LEMMA 4.10. *If $F[M, N, k+1, \ell]$ has one external gap at r (note $r = M$ or $r = N$) and no other gaps, then*

$$F[M, N, 2k+1, \ell+1] = \left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup s_{r,\ell+1}.$$

Proof. Without loss of generality, $r = M$. Assume that the equality in the statement of the lemma is not true. Since the right-hand side of that inequality is a level-($\ell+1$) $(2k+1)$-forest over $[M, N]$ that is not optimal, we have

$$\left|\left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup s_{M,\ell+1}\right| > |F[M, N, 2k+1, \ell+1]|.$$

Thus,

$$|F[M, N, k+1, \ell]| > \left|\left(F[M, N, 2k+1, \ell+1] \cup B^{\ell}\right) - s_{M,\ell+1}\right|.$$

If $F[M, N, 2k+1, \ell+1]$ has a gap at M, then $\left(F[M, N, 2k+1, \ell+1] \cup B^{\ell}\right) - s_{M,\ell+1}$ is a $(k+1)$-forest over $[M, N]$, which is a contradiction, since it cannot have smaller weight than the optimal forest. Otherwise, by Lemma 4.9, there exists a level-($\ell+1$) $(2k+2)$-forest F' over $[M, N]$ such that

$$\left|F[M, N, 2k+1, \ell+1] - s_{M,\ell+1}\right| > \left|F'\right|.$$

Therefore

$$|F[M, N, k+1, \ell]| > \left|F' \cup B^{\ell}_{[M,N]}\right|$$

which is a contradiction, since $F' \cup B^{\ell}_{[M,N]}$ is a $(k+1)$-forest over $[M, N]$, which cannot have smaller weight than the optimal forest. □

LEMMA 4.11. *If $F[M, N, k+1, \ell]$ has two external gaps and no internal gap then*

$$\left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup s_{M,\ell+1} \cup s_{N,\ell+1} = F[M, N, 2k, \ell+1].$$

Proof. Assume that the equality in the statement of the lemma is not true. Since the left-hand side of that inequality is a level-$(\ell+1)$ $2k$-forest over $[M, N]$ that is not optimal, we have

$$\left|\left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup s_{M,\ell+1} \cup s_{N,\ell+1}\right| > |F[M, N, 2k, \ell+1]| \,.$$

Thus,

$$|(F[M, N, k+1, \ell])| > \left|\left(F[M, N, 2k, \ell+1] - s_{M,\ell+1} - s_{N,\ell+1}\right) \cup B^{\ell}_{[M,N]}\right| .$$

If $F = F[M, N, 2k, \ell+1]$ has gaps at both M and N, then

$$\left(F[M, N, 2k, \ell+1] - s_{M,\ell+1} - s_{N,\ell+1}\right) \cup B^{\ell}_{[M,N]}$$

is a $(k+1)$-forest over $[M, N]$, which is a contradiction, since it cannot have smaller weight than the optimal forest. Otherwise, without loss of generality, F has no gap at M. By Lemma 4.9, there exists a level-$(\ell+1)$ $(2k+1)$-forest F' over $[M, N]$ such that $\left|F - s_{M,\ell+1}\right| > \left|F'\right|$.

We now consider two cases, depending on whether F' has a gap at N. Suppose F' has a gap at N. Then $\left|\left(F' - s_{N,\ell+1}\right) \cup B^{\ell}_{[M,N]}\right| < |F[M, N, k+1, \ell]|$ which is a contradiction, since $\left(F' - s_{N,\ell+1}\right) \cup B^{\ell}_{[M,N]}$ is a $(k+1)$-forest over $[M, N]$, which cannot have smaller weight than the optimal forest.

Suppose F' has no gap at N. By Lemma 4.9, there exists a level-$(\ell+1)$ $(2k+2)$-forest F'' over $[M, N]$ such that $\left|F' - s_{N,\ell+1}\right| > \left|F''\right|$. Then $\left|F'' \cup B^{\ell}_{[M,N]}\right| < |F[M, N, k+1, \ell]|$ which is a contradiction, since $F'' \cup B^{\ell}_{[M,N]}$ is a $(k+1)$-forest over $[M, N]$, which cannot have smaller weight than the optimal forest.

This concludes the proof of Lemma 4.11. □

Using Lemma 4.4, we can summarize the results of Lemmas 4.8, 4.10, and 4.11 in the following corollary.

LEMMA 4.12. *If $F[M, N, k+1, \ell]$ does not have internal gaps then*

$$\left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup B^{(\ell+1)}_{[M,N]}$$

and

$$\left(F[M, N, k, \ell] - B^{\ell}\right) \cup B^{(\ell+1)}_{[M,N]}$$

are optimal level-$(\ell+1)$ forests.

Now we are ready to conclude the proof of the inclusion property.

Proof (of Theorem 4.7). As in the proof of the Theorem 3.10, we prove the inclusion property by backwards induction on the level, ℓ, and within each level by induction on the base length. If $\ell = L$, then $F[M, N, M-N+1, L]$ is the only level-L forest over $[M, N]$. The only forests of base length 1 are singleton tiles. Thus, for $\ell = L$ or $N = M$, the theorem holds.

Assume that the theorem holds for all $\ell' > \ell$ and, within ℓ, for all $[M', N']$ such that $N' - M' < N - M$. We consider four cases depending on relative shapes of $F[M, N, k+1, \ell]$ and $F[M, N, k, \ell]$.

Case 1. $F[M, N, k, \ell]$ and $F[M, N, k+1, \ell]$ have a common internal cut, c.

In this case, by Lemma 4.6, $F[M, N, k, \ell]$ and $F[M, N, k+1, \ell]$ are identical on one side, say the left side, of the cut. Thus, for some k',

$$F[M, N, k, \ell]_{[c+1,N]} = F[c+1, N, k', \ell]$$

and

$$F[M, N, k+1, \ell]_{[c+1,N]} = F[c+1, N, k'+1, \ell].$$

By the inductive hypothesis,

$$F[c+1, N, k', \ell] = F[c+1, N, k'+1, \ell] \cup p,$$

where p is a level-ℓ glue-pack with respect to $F[c+1, N, k'+1, \ell]$. Thus p is also a level-ℓ glue-pack with respect to $F[M, N, k+1, \ell]$. Furthermore,

$$F[M, N, k, \ell] = F[M, N, k+1, \ell] \cup p.$$

Case 2. $F[M, N, k+1, \ell]$ does not have a gap.

From Lemma 4.4 it follows that $F[M, N, k, \ell]$ also does not have a gap. By Lemma 4.8 we have

$$F[M, N, k+1, \ell] - B^{\ell} = F[M, N, 2k+2, \ell+1]$$

and

$$F[M, N, k, \ell] - B^{\ell} = F[M, N, 2k, \ell+1].$$

By the inductive hypothesis we have

$$F[M, N, 2k, \ell+1] = F[M, N, 2k+2, \ell+1] \cup p_1 \cup p_2$$

where p_1 and p_2 are level-$(\ell+1)$ items such that p_1 is a glue-pack with respect to $F[M, N, 2k+2, \ell+1]$ and p_2 is a glue-pack with respect to $F[M, N, 2k+1, \ell+1] \cup p_1$. Since $F[M, N, k+1, \ell]$ does not have gaps, p_1 is a level-$(\ell+1)$ glue-pack with respect to $F[M, N, k+1, \ell]$, p_2 is a level-$(\ell+1)$ glue-pack with respect to $F[M, N, k+1, \ell] \cup p_1$, and $p_1 \cup p_2$ is level-ℓ connected with respect to $F[M, N, k+1, \ell]$. Therefore

$$F[M, N, k, \ell] - F[M, N, k+1, \ell] = p_1 \cup p_2$$

and $p_1 \cup p_2$ is a glue-pack with respect to $F[M, N, k+1, \ell]$.

Case 3. $F[M, N, k+1, \ell]$ has exactly one external gap.

By symmetry, we may assume that the one external gap of $F[M, N, k+1, \ell]$ is at M.

We first show that we may assume that $F[M, N, k+1, \ell]$ has only one gap and that $F[M, N, k, \ell]$ has no gaps. If $F[M, N, k+1, \ell]$ has an internal gap, then we can reduce to Case 1 by Lemma 4.5. If $F[M, N, k, \ell]$ has a gap at M, then M is a common internal cut, and we again reduce to Case 1. By Lemma 4.4, $F[M, N, k, \ell]$ cannot otherwise have a gap.

By Lemma 4.8 we have

$$F[M, N, k, \ell] - B^{\ell} = F[M, N, 2k, \ell+1]$$

and by Lemma 4.10 we have

$$\left(F[M, N, k+1, \ell] - B^{\ell}\right) \cup s_{r,\ell+1} = F[M, N, 2k+1, \ell+1].$$

Thus

$$F[M, N, k, \ell] = F[M, N, k+1, \ell] \cup s_{r,\ell+1} \cup p_2,$$

where $p_2 = F[M, N, 2k, \ell + 1] - F[M, N, 2k + 1, \ell + 1]$ is a level-$(\ell + 1)$ glue-pack with respect to $F[M, N, 2k + 1, \ell + 1]$. Thus p_2 is a level-$(\ell + 1)$ glue-pack with respect to $F[M, N, k + 1, \ell] \cup s_{r,\ell+1}$. Since $F[M, N, k + 1, \ell]$ does not have an internal gap, $p = p_2 \cup s_{r,\ell+1}$ is level-ℓ connected with respect to $F[M, N, k + 1, \ell]$ and therefore is a level-ℓ glue-pack with respect to $F[M, N, k + 1, \ell]$.

Case 4. $F[M, N, k + 1, \ell]$ has two external gaps.

As in the previous case, we assume that $F[M, N, k + 1, \ell]$ does not have internal gaps and that $F[M, N, k, \ell]$ does not have any gaps, since otherwise the problem can be reduced to Case 1. By Lemma 4.8 we have

$$F[M, N, k, \ell] - B^{\ell} = F[M, N, 2k, \ell + 1]$$

and by Lemma 4.11 we have

$$\left(F[M, N, k + 1, \ell] - B^{\ell}\right) \cup s_{M,\ell+1} \cup s_{N,\ell+1} = F[M, N, 2k, \ell + 1].$$

Thus

$$F[M, N, k, \ell] = F[M, N, k + 1, \ell] \cup s_{M,\ell+1} \cup s_{N,\ell+1}.$$

Note that $s_{M,\ell+1} \cup s_{N,\ell+1}$ is a glue-pack with respect to $F[M, N, k + 1, \ell]$. The result follows.

This concludes the proof of Theorem 4.7. □

4.3. The Package Search algorithm. Theorem 4.7 gives us the first hint leading to an algorithm for construction of an optimal alphabetic height-restricted tree. Namely, we need to find the smallest weight package that is a level-0 glue-pack with respect to B^0. This gives us the optimal height-restricted $(n - 1)$-forest. Then we find the smallest weight package that is a glue-pack with respect to that forest. This leaves us with the optimal height-restricted $(n - 2)$-forest. We repeat this step until we construct the optimal tree.

Theorem 4.7 gives rise to the following definition.

DEFINITION. *For any M, N, r, ℓ such that*

$$1 \leq M < N \leq n,\ 0 \leq \ell \leq L,\ \textit{and}\ 1 \leq r \leq N - M - \lceil (N - M + 1)2^{\ell - L} \rceil,$$

define the optimal level-ℓ package of rank r over $[M, N]$, $p^{\ell}_{[M,N]}(r)$, to be the difference

$$p^{\ell}_{[M,N]}(r) = F[M, N, N - M - r + 1, \ell] - F[M, N, N - M - r + 2, \ell].$$

An optimal level-ℓ item is defined to be either an optimal level-ℓ package or a level-ℓ tile.

With the above definition, the algorithm described informally at the beginning of this section can be described as follows:

ALGORITHM 1. **Package Search (L)**

```
1. $F^{\ell} := B^{\ell}$;
2. for $r := 1$ to $n - \lceil n2^{\ell-L} \rceil - 1$ do
   2.1. find $p^{\ell}_{[1,n]}(r)$;
   2.2. $F^{\ell} := F^{\ell} \cup p^{\ell}_{[1,n]}(r)$.
```

The most difficult step of the above algorithm is step 2.1, in which we find the optimal level-ℓ package of rank r, $p^{\ell}_{[1,n]}(r)$. In the next section we will examine properties of optimal packages that lead to an efficient implementation of this step. In fact, we shall find that optimal level-0 packages over the interval $[1, n]$ can be constructed recursively from optimal packages of lower levels over the same interval. Consequently, in our implementation we use only optimal packages over interval $[1, n]$. However we shall find the more general definition given in this section convenient in the inductive proofs of basic properties of optimal packages.

5. Monotonicity and decomposition properties of optimal packages. In this section, we continue to assume that the uniqueness hypothesis of §4 holds.

The basic difficulty of the Package Search algorithm presented in the previous section is to describe an efficient implementation of step 2.1, in which we find the optimal level-ℓ package, $p^{\ell}_{[1,n]}(r)$, i.e., the smallest weight level-ℓ package that forms a glue-pack with respect to F^{ℓ}. Thus, we need more information about the structure of optimal glue-packs. Their basic properties are given in the theorem below.

THEOREM 5.1 (main theorem). *Given M, N, r, ℓ, if $1 \leq M < N \leq n$, $0 \leq \ell \leq L$, and $1 \leq r \leq N - M - \lceil (N - M + 1)2^{\ell - L} \rceil$, then we have the following items.*

1. *The decomposition property*:

   1.1 *$p^{\ell}_{[M,N]}(r)$ is the smallest weight union of two level-$(\ell+1)$ items that are optimal over $[M, N]$ which is level-ℓ connected with respect to $F[M, N, N - M - r + 2, \ell]$.*

   1.2 *$p = p^{\ell}_{[M,N]}(r)$ has a normal parse tree with both children of p being optimal level-$(\ell + 1)$ items and the right child being heavier than the left child.*

2. *The monotonicity property*:

   If $r > 1$, then $\left|p^{\ell}_{[M,N]}(r-1)\right| < \left|p^{\ell}_{[M,N]}(r)\right|$.

3. *The package span property*:

   Write $p = p^{\ell}_{[M,N]}(r)$. If $i \in [lboundary(p), rboundary(p)]$ then $|p| > \left|s_{i,\ell+1}\right|$.

Proof. The proof is by triple induction: within an interval on the rank, r, of an optimal level-ℓ package, within the level, on the base length of the interval, and backwards induction on the level ℓ.

Formally, our inductive hypothesis is that the statement of the theorem holds for any of the following three "predecessor" situations:

(i) The same values for $[M, N]$ and ℓ, and a smaller value for r.

(ii) The same value for ℓ, a proper subinterval $[M', N']$ for $[M, N]$, and any value for r.

(iii) If $\ell < L$: $(\ell + 1)$ for ℓ, and any values for $[M, N]$ and r.

For $\ell = L$, the result follows vacuously from the fact that there are no level-L packages.

Assuming the inductive hypothesis, we first prove the decomposition property, then the monotonicity property, and finally the package span property.

The following lemma will be used in the proof of the decomposition property and the monotonicity property.

LEMMA 5.2. *Suppose Q is a skyline, and that p and q are level-$(\ell + 1)$ items which are optimal over $[M, N]$ such that q is a glue-pack with respect to Q, p is a glue-pack with respect to $Q \cup q$, and $|p| < |q|$. Then p is a glue-pack with respect to Q, and q is a glue-pack with respect to $Q \cup p$.*

Proof. The second part of the conclusion follows immediately from Lemma 3.5. Thus, we need only show that p is a glue-pack with respect to Q. If p is a tile, this is trivial, so assume p is a package. The remainder of the proof is in two cases, depending on whether q is a tile or a package.

Suppose q is a tile, say, $s_{i,\ell+1}$. By Lemma 3.9, p cannot span i, since otherwise it must contain $s_{i,\ell+2}$, which is at least as heavy as q. Without loss of generality, *lboundary* $(p) > i$. Applying Lemma 3.7 twice, we see that p is a glue-pack with respect to $(Q \cup q)_{[i+1,N]} = Q_{[i+1,N]}$, and thus also with respect to Q.

Suppose q is a package. Let $F = F[M, N, N - M - t + 2, \ell + 1]$, where t is that rank for which $p = p^{\ell+1}_{[M,N]}(t)$. By the definition of optimal package, p is a glue-pack with respect to F. By monotonicity at level $(\ell + 1)$, which follows from part (iii) of the inductive hypothesis, $q = p^{\ell+1}_{[M,N]}(t')$ for some $t' > t$, hence $q \cap F = \emptyset$. By Lemma 3.6, p is a glue-pack with respect to $F \cap (Q \cup q) = F \cap Q$. By Lemma 3.5, p is a glue-pack with respect to Q. □

5.1. The proof of the decomposition property. Let $p = p^{\ell}_{[M,N]}(r)$ for $\ell < L$, and let p_1, p_2 be the children of p in its normal parse tree. We need only prove that p_1 and p_2 can be chosen to be level-$(\ell + 1)$ optimal items over $[M, N]$, since the other parts of the decomposition property then follow by the definition of $p^{\ell}_{[M,N]}(r)$ and by Lemma 5.2. If both p_1 and p_2 are tiles, the decomposition property follows immediately. Thus, we can assume that at least one of them, namely p_2, is a level-$(\ell + 1)$ package.

By the definition of an optimal level-ℓ package over $[M, N]$ of rank r, we have

$$p = F[M, N, N - M - r + 1] - F[M, N, N - M - r + 2].$$

Let i be the largest integer such that i is a gap of $F[M, N, N - M - r + 2]$ and $i \leq$ *lboundary* (p). Let $i = M$ if such a gap does not exist. Let j be the smallest integer such that j is a gap of $F[M, N, N - M - r + 2]$ and $j \geq rboundary(p)$. Let $j = N$ if such a gap does not exist.

We note that p is an optimal package over $[i, j]$. If q were a better choice of level-ℓ glue-pack over $[i, j]$, then, by Lemma 3.7, q would also be a better choice of level-ℓ glue-pack over $[M, N]$, so p would not be optimal over $[M, N]$.

We consider two cases: when $[i, j] = [M, N]$ and $[i, j] \neq [M, N]$.

If $[i, j] = [M, N]$ then, by the definition of i and j, $F[M, N, N - M - r + 2]$ has no internal gaps. Thus, by Corollary 4.12,

$$F_1 = \left(F[M, N, N - M - r + 2, \ell] - B^{\ell}\right) \cup B^{\ell+1}_{[M,N]}$$

and

$$F_2 = \left(F[M, N, N - M - r + 1, \ell] - B^{\ell}\right) \cup B^{\ell+1}_{[M,N]}$$

are optimal forests over $[M, N]$. If neither p_1 nor p_2 is a tile, then

$$p = p_1 \cup p_2 = F_2 - F_1.$$

By the definition of an optimal package, F_2 is obtained from F_1 by adding two level-$(\ell + 1)$ optimal packages over $[M, N]$. Thus, p_1 and p_2 can be chosen to be optimal over $[M, N]$, and we are done.

If p_1 is a tile, then

$$p_2 = F_2 - F_1.$$

Thus, by the definition of an optimal package, p_2 is an optimal level-$(\ell + 1)$ package over $[M, N]$, and we are done.

This concludes the proof of the decomposition property for the case when $[M, N] = [i, j]$.

Assume now that $[i, j]$ is a proper subinterval of $[M, N]$. Let

$$F = \left(F[M, N, N - M - r + 2, \ell] - B^{\ell}\right) \cup B^{\ell+1}_{[M,N]}.$$

By part (i) of the inductive hypothesis, F is a union of level-$(\ell + 1)$ items which are optimal over $[M, N]$.

By Lemmas 4.2, 4.4, and 4.12, since $F[M, N, N - M - r + 2, \ell]_{[i,j]}$ does not have internal gaps, $F_{[i,j]}$ and $F_{[i,j]} \cup p$ are optimal $(\ell + 1)$-forests over $[i, j]$.

Thus, if p_1 is a tile, then $F_{[i,j]} \cup p = F_{[i,j]} \cup p_2$, hence p_2 is optimal over $[i, j]$. If p_1 is a package, then p_1, p_2 can be chosen to be optimal over $[i, j]$. By Lemma 5.2, $|p_1| < |p_2|$. For $i = 1, 2$, let P_i be the union of all level-$(\ell + 1)$ packages of weight less than $|p_i|$ which are optimal over $[M, N]$. Let $F_i = P_i \cup B^{\ell+1}_{[M,N]}$, which is an optimal level-$(\ell + 1)$ forest, by monotonicity at level $(\ell + 1)$. We will prove the following two claims:

Claim 1. If p_1 is a package, then $(F_1)_{[i,j]} = F_{[i,j]}$.

Claim 2. If p_1 is an optimal item over $[M, N]$, then $(F_2)_{[i,j]} = F_{[i,j]} \cup p_1$.

We conclude the proof of the decomposition property, assuming Claims 1 and 2.

If p_1 is a tile, it is optimal. If p_1 is a package, let q_1 be the minimal level-$(\ell+1)$ glue-pack with respect to F_1. By definition of optimal package, q_1 is an optimal level-$(\ell + 1)$ package over $[M, N]$, and $|q_1| \geq |p_1|$, since otherwise $q_1 \subseteq P_1$. By Lemma 3.7 and Claim 1, p_1 is a glue-pack with respect to F_1, thus $|p_1| \geq |q_1|$, since otherwise q_1 would not be minimal. By the uniqueness hypothesis, $p_1 = q_1$. Thus, p_1 is an optimal level-$(\ell + 1)$ package over $[M, N]$.

Let q_2 be the minimal level-$(\ell+1)$ glue-pack with respect to F_2. By definition of optimal package, q_2 is optimal over $[M, N]$, and $|q_2| \geq |p_2|$, since otherwise $q_2 \subseteq P_2$. By Lemma 3.7 and Claim 2, p_2 is a glue-pack with respect to F_2, thus $|p_2| \geq |q_2|$, since otherwise q_2 would not be minimal. By the uniqueness hypothesis, $p_2 = q_2$. Thus, p_2 is an optimal level-$(\ell + 1)$ package over $[M, N]$.

Thus, the decomposition property holds if Claims 1 and 2 are true. It remains only to prove those claims.

We will use the following notation:

We define a COS (COS = *Component Of Subforest*) *over* $[M', N']$ to be an optimal level-$(\ell+1)$ forest over a subinterval $[M', N'] \subseteq [M, N]$ which is the disjoint union of level-$(\ell+1)$ items which are optimal over $[M, N]$.

In particular, if either $M' = M$ or M' is a gap of $F[M, N, N - M - r + 2, \ell]$, and if either $N' = N$ or N' is a gap of $F[M, N, N - M - r + 2, \ell]$, then $F_{[M',N']}$ is a COS.

LEMMA 5.3. *If Q is a* COS *over $[M', N'] \subseteq [M, N]$, then those level-$(\ell + 1)$ items in Q which are optimal over $[M, N]$ are also optimal items over $[M', N']$.*

Proof. We only need to consider packages, since tiles are automatically optimal items. Write $Q = B^{\ell+1}_{[M',N']} \cup q_1 \cup q_2 \cup \cdots \cup q_m$, where each q_t is a level-$(\ell + 1)$ package which is optimal over $[M, N]$, and where $|q_t| < |q_{t+1}|$. We prove $q_t = p^{\ell+1}_{[M',N']}(t)$ for each t. Suppose this is not the case. Pick the smallest t for which $q_t \neq q'_t = p^{\ell+1}_{[M',N']}(t)$. Since Q is the optimal $(N' - M' - m + 1)$-forest at level $(\ell+1)$ over $[M', N']$, it contains $p^{\ell+1}_{[M',N']}(t')$ for all $t' \in [1, m]$. Hence, in particular, it contains q'_t. Let F_t be the level-$(\ell + 1)$ forest over $[M, N]$ consisting of $B^{\ell+1}_{[M,N]}$ together with all level-$(\ell + 1)$ packages of weight less than $|q_t|$ which are optimal over $[M, N]$. By monotonicity at level $(\ell + 1)$, F_t is an optimal forest over $[M, N]$, and q_t is the smallest level-$(\ell + 1)$ glue-pack with respect to F_t.

Let $H = B^{\ell+1}_{[M',N']} \cup q_1 \cup \cdots \cup q_{t-1} = F[M', N', N' - M' - t + 2, \ell + 1]$, an optimal level-$(\ell + 1)$ forest over $[M', N']$. Since q'_t is disjoint from $H = F_t \cap Q$ and is a subset of Q, it is disjoint from F_t. Since q'_t is level-$(\ell + 1)$ connected with respect to H, it is level-$(\ell + 1)$ connected with respect to F_t, since $H \subseteq F_t$. Thus, q'_t is a level-$(\ell + 1)$ glue-pack with respect to F_t, and by optimality of q_t and uniqueness, $|q'_t| > |q_t|$.

On the other hand, q_t is level-$(\ell + 1)$ connected with respect to both F_t and Q, hence also with respect to H, by Remark 3.3. Since q_t is disjoint from H, it is a glue-pack with respect to H, contradicting the minimality of q'_t. □

Let Q be a COS over $[M', N']$, and q' be the smallest weight level-$(\ell + 1)$ glue-pack with respect to Q. Note that q' is optimal over $[M', N']$. Define $SE(Q)$ to be the union of all level-$(\ell+1)$ packages, x, which are optimal over $[M, N]$ and satisfy the following conditions:

(i) $|x| \leq |q'|$;

(ii) $x \cap Q = \emptyset$;

(iii) $x_{[M',N']} \neq \emptyset$.

Define $S(Q)$ to be to be the union of all level-$(\ell+1)$ packages, x, which are optimal over $[M, N]$ and satisfy the following conditions:

(i) $|x| < |q'|$;
(ii) $x \cap Q = \emptyset$;
(iii) $x_{[M',N']} \neq \emptyset$.

Define $G(Q)$ to be the union of all level-$(\ell+1)$ packages, x, which are optimal over $[M, N]$ and satisfy the following conditions:

(i) $|x| > |q'|$;
(ii) $x \subseteq Q$.

LEMMA 5.4. *If Q is a* COS *over $[M', N'] \subseteq [M, N]$, q' is the smallest weight level-$(\ell+1)$ glue-pack with respect to Q, and q is the least weight member of $SE(Q)$, then $q_{[M',N']} = q$ implies that $q = q'$.*

Proof. If $[M', N'] = [M, N]$, then the lemma is trivial since q' is optimal over $[M', N'] = [M, N]$. Thus, we can assume $[M', N']$ is a proper subinterval of $[M, N]$. Let F_q be the level-$(\ell+1)$ forest over $[M, N]$ consisting of $B^{\ell+1}_{[M,N]}$ together with all level-$(\ell+1)$ packages of weight less than $|q|$ which are optimal over $[M, N]$. By monotonicity at level $(\ell+1)$, F_q is an optimal forest over $[M, N]$, and q is the smallest level-$(\ell+1)$ glue-pack with respect to F_q.

By Lemma 5.3, we can write $Q = B^{\ell}_{[M',N']} \cup q_1 \cup \cdots \cup q_m$, where each q_t is a level-$(\ell+1)$ package which is optimal over both $[M, N]$ and $[M', N']$, and $q_t = p^{\ell+1}_{[M',N']}(t)$. Note $q' = p^{\ell+1}_{[M',N']}(m+1)$, and thus $|q_t| < |q'|$ for all t, by the monotonicity property over $[M', N']$ implied by part (iii) of the inductive hypothesis.

Let $Q' = (F_q)_{[M',N']}$. By the minimality of q, $Q' \subseteq Q$. Since q is the smallest weight glue-pack with respect to F_q and $q_{[M',N']} = q$, it follows that q is a glue-pack with respect to Q', by Lemma 3.7. Since $Q' \subseteq Q$, by Remark 3.3, we know q is ℓ-connected with respect to Q. Since $q \cap Q = \emptyset$, q is a glue-pack with respect to Q. Therefore $|q| \geq |q'|$. By definition of $SE(Q)$, $|q| \leq |q'|$. By the uniqueness hypothesis, $q' = q$. □

Since $F_{[i,j]}$ is a COS over $[i, j]$, we can rewrite Claim 1 as follows.

Claim 1′. If p_1 is a package then $S(F_{[i,j]}) = \emptyset$ and $G(F_{[i,j]}) = \emptyset$.

Since p_1 is optimal over $[i, j]$, and since $F_{[i,j]} \cup p_1$ is a COS over $[i, j]$ if p_1 is optimal over $[M, N]$, we can rewrite Claim 2 as follows:

Claim 2′. If p_1 is an optimal item over $[M, N]$, then $S(F_{[i,j]} \cup p_1) = \emptyset$ and $G(F_{[i,j]} \cup p_1) = \emptyset$.

LEMMA 5.5. *Suppose Q is a* COS *over $[M', N'] \subseteq [M, N]$. Then $G(Q) = \emptyset$.*

Proof. By Lemma 5.3, $Q = B^{\ell+1}_{[M',N']} \cup q_1 \cup \cdots \cup q_m$, where $q_1, \ldots q_m$ are level-$(\ell+1)$ items, each of which is optimal over both $[M, N]$ and $[i, j]$. Let q be the smallest weight glue-pack with respect to Q. By the monotonicity property at level $(\ell+1)$ over $[i, j]$ implied by part (iii) of the inductive hypothesis, all q_t have weight less than $|q|$. Equivalently, $G(Q) = \emptyset$. □

By Lemma 5.5, Claims 1′ and 2′ are equivalent to the following two claims.

Claim 1″. If p_1 is a package, then $S(F_{[i,j]}) = \emptyset$.

Claim 2″. If p_1 is an optimal item over $[M, N]$, then $S(F_{[i,j]} \cup p_1) = \emptyset$.

We now finish the proof of Theorem 5.1 by proving Claims 1″ and 2″. First note that Lemma 5.4 immediately implies the following corollary.

COROLLARY 5.6. *Let Q be a* COS *over $[M', N'] \subseteq [M, N]$. Let q be the smallest weight member of $SE(Q)$, if one exists. Then $q_{[M',N']} = q$ implies $S(Q) = \emptyset$.*

We break the argument into two cases: when p_1 is a tile and p_2 is a package, and when both p_1 and p_2 are packages.

Case 1. Consider first the case when p_1 is a tile. In this case we need to prove only Claim 2$''$. By the gap-filling property (Lemma 3.9), $p_1 = s_{i,\ell+1}$ or $p_1 = s_{j,\ell+1}$. Assume, without loss of generality, that the first case holds. Note that $F = F \cup p_1$ and $F_{[i,j]} = F_{[i,j]} \cup p_1$. Suppose $S(F_{[i,j]}) \neq \emptyset$. Let q be the smallest weight member of $S(F_{[i,j]})$. By Corollary 5.6, $q_{[i,j]} \neq q$. Thus q must span an external gap of $F_{[i,j]}$, either i or j. If q spans j then, by the package span property on level $(\ell + 1)$, implied by part (iii) of the inductive hypothesis, $|s_{j,\ell+2}| < |q|$. Thus, by the monotonicity property of the weight function, $|s_{j,\ell+1}| < |q|$. By minimality of q', $|q| < |p_2|$. Thus $p_1 \cup s_{j,\ell+1} = s_{i,\ell+1} \cup s_{j,\ell+1}$ is a level-ℓ glue-pack with respect to $F[M, N, N - M - r + 2, \ell]$ of weight less than $|p|$, contradicting the minimality of p.

Thus we can assume that q spans i. Let k be the greatest integer smaller than i such that $F[M, N, N - M - r + 2, \ell]$ has a gap at k. If no such gap exists, let $k = M$.

Note that if k is a gap of $F[M, N, N - M - r + 2, \ell]$, then there does not exist an optimal package over $[M, N]$ of weight less than or equal to $|q|$ that spans k, since otherwise, by the package span property at level $(\ell + 1)$, and by the monotonicity property of the weight function, $s_{k,\ell+1} \cup s_{i,\ell+1}$ is a glue-pack with respect to $F[M, N, N - M - r + 2, \ell]$ of weight less than $|p|$, contradicting the minimality of p.

By Lemmas 4.2, 4.12, and part (ii) of the inductive hypothesis, $F_{[k,i]}$ is a COS over $[k, i]$.

Let q' be the smallest weight level-$(\ell + 1)$ package optimal over $[M, N]$ not in F such that $|q'| \leq |q|$ and $q'_{[k,j]} \neq \emptyset$. The existence of q' follows from the existence of q. Since q is a minimal weight package of $S(F_{[i,j]})$, *rboundary*$(q') < i$ (otherwise $q' \neq q$ and $q' \subseteq S(F_{[i,j]})$). Since q' does not span k, *lboundary*$(q') > k$. Thus $q'_{[k,i]} = q'$. By Lemma 5.4, q' is a glue pack with respect to $F_{[k,i]}$. Therefore $s_{i,\ell+1} \cup q'$ is a glue-pack with respect to $F[M, N, N - M - r + 2, \ell]$ of weight less than $|p|$, contradicting the minimality of p.

Case 2. Assume now that both p_1 and p_2 are level-$(\ell + 1)$ packages over $[M, N]$. We need to prove both Claim 1$''$ and Claim 2$''$. First we prove Claim 1$''$, i.e., that $S(F_{[i,j]}) = \emptyset$. Assume $S(F_{[i,j]}) \neq \emptyset$. Let q be the smallest weight member of $S(F_{[i,j]})$. By Corollary 5.6, $q_{[i,j]} \neq q$. Thus q must span an external gap of $F_{[i,j]}$ at i or j. Without loss of generality, q spans i. By the package span property at level $(\ell + 1)$, $|q| > |s_{i,\ell+2}| \geq |s_{i,\ell+1}|$. Thus $|p_1| > |s_{i,\ell+1}|$ and therefore $|p_2| > |s_{i,\ell+1}|$. This implies that $s_{i,\ell+1} \cup p_1$ is a glue-pack with respect to $F_{[i,j]}$ of weight less than $|p|$, contradicting the minimality of p.

It remains to prove Claim 2$''$ for the case when p_1, p_2 are both level-$(\ell + 1)$ packages and under the assumption that Claim 1 is true. Assume that the claim is false, i.e., $S(F_{[i,j]} \cup p_1) \neq \emptyset$. Let q be the smallest weight optimal level-$(\ell + 1)$ package over $[M, N]$ in $S(F_{[i,j]} \cup p_1)$. By Corollary 5.6, $q_{[i,j]} \neq q$, and therefore q must span an external gap of $F_{[i,j]}$ at i or j. Without loss of generality, q spans i. By the package span property at level $(\ell + 1)$, $|q| > |s_{i,\ell+1}| \geq |s_{i,\ell+1}|$, hence $|p_2| > |s_{i,\ell+1}|$. But this implies that $s_{i,\ell+1} \cup p_1$ is a glue-pack with respect to $F_{[i,j]}$ of weight less than $|p|$, contradicting the minimality of p.

This finishes the proof of the decomposition property.

5.2. The proof of the monotonicity property. Let $F = F[M, N, N - M - r + 3]$. Let $q = p^{\ell}_{[M,N]}(r - 1)$, and $p = p^{\ell}_{[M,N]}(r)$, and suppose that $|q| \geq |p|$. By the uniqueness hypothesis, $|q| > |p|$. Let p_1, p_2 be the children of p in its normal parse tree, and let q_1, q_2 be the children of q in its normal parse tree. By the decomposition property, and by Lemma 5.2, we may assume that p_1 is the left child of p, q_1 is the left child of q, $|q_1| < |q_2|$, and $|p_1| < |p_2|$. Thus, $|p_1| < |q_2|$.

If $q_1 \cup p_1$ were a glue-pack for F, it would be lighter than q, contradicting the minimality of q. Thus, it cannot be a glue-pack. But we know that q_1 is a glue-pack with respect to F, since q is a glue-pack with respect to F. We also know that p_1 is a glue-pack with respect to $F \cup q_1$, by Lemma 5.2, since $|p_1| < |q_2|$ and p_1 is a glue-pack with respect to $F \cup q_1 \cup q_2$.

Thus, there must be a gap of F that separates q_1 from p_1, say at i. Without loss of generality, *rboundary* $(p_1) > i$ and *lboundary* $(q_1) < i$.

By Lemma 3.9, either $q_2 = s_{i,\ell+1}$ or *rboundary* $(q_2) < i$.

Case 1. Suppose *rboundary* $(q_2) < i$. By Lemma 3.9 and the fact that p_1 lies entirely to the right of i, we have *lboundary* $(p) \geq i$. Thus, $p_{[i,N]} = p$. Applying Lemma 3.7 twice, we see that p is a glue-pack with respect to $(F \cup q)_{[i,N]} = F_{[i,N]}$, and thus also with respect to F, contradicting the minimality of q.

Case 2. Suppose $q_2 = s_{i,\ell+1}$. If *lboundary* $(p_2) > i$, then $p_{[i+1,N]} = p$. Applying Lemma 3.7 twice, we see that p is a glue-pack with respect to $(F \cup q)_{[i+1,N]} = F_{[i+1,N]}$, and thus also with respect to F, contradicting the minimality of q. Thus, *lboundary* $(p_2) \leq i$.

Recall that p_1 is a glue-pack with respect to $F \cup q$. Recall also that $(p_1)_{[i,N]} = p_1$. Applying Lemma 3.7 twice, we see that p_1 is a glue-pack with respect to $(F \cup q)_{[i,N]} = F_{[i,N]} \cup q_2$, and thus also with respect to $F \cup q_2$. Since it is a tile, q_2 is a glue-pack with respect to F. If q_2 and p_1 are not level-ℓ connected with respect to F, they must be separated by a gap, say at j, where $i < j <$ *lboundary* (p_1). By Lemma 3.9, *lboundary* $(p) \geq j$, hence *lboundary* $(p_2) \geq j > i$, contradiction. Thus, $q_2 \cup p_1$ is a glue-pack with respect to F, which must be heavier than q, since q is minimal. It follows that $|q_1| < |p_1|$. From the fact that $|p| < |q|$, we can then conclude that $|p_2| < |q_2|$.

Now p_2 is a level-$(\ell+1)$ glue-pack with respect to $F \cup q \cup p_1$, which does not contain $s_{i,\ell+2}$. By Lemma 3.9, p_2 cannot span i, since then it would contain $s_{i,\ell+2}$, which is at least as heavy as q_2 by monotonicity of weights. The only remaining possibility is that *rboundary* $(p_2) < i$, i.e., $(p_2)_{[M,i-1]} = p_2$. Applying Lemma 3.7 twice, we see that p_2 is a glue-pack with respect to $(F \cup q \cup p_1)_{[M,i-1]} = (F \cup q_1)_{[M,i-1]}$, and thus also with respect to $F \cup q_1$.

If q_1 and p_2 were separated by a gap of F, say j, then $j < i$, since both q_1 and p_2 extend to the left of i. If q_1 is to the left of j, then that gap separates q_1 from q_2, contradicting the fact that q is a glue-pack of F, while if p_2 is to the left of j, then that gap (which is still a gap of $F \cup q$) separates p_2 from p_1, contradicting the fact that p is a glue-pack of $F \cup q$. Thus, $q_1 \cup p_2$ is a glue-pack with respect to F. By minimality of q, $|q_2| < |p_2|$, contradicting the fact that $|p_2| < |q_2|$.

This finishes the proof of the monotonicity property.

5.3. The proof of the package span property. Suppose $i \in [lboundary(p), rboundary(p)]$. If $F[M, N, N - M - r + 2, \ell]$ has a gap at i, then the result follows from Lemma 3.9. Otherwise, $s_{i,\ell+1} \in p^{\ell}_{[M,N]}(t)$ for some $t < r$, and $\left|s_{i,\ell+1}\right| < \left|p^{\ell}_{[M,N]}(t)\right| < |p|$ by the monotonicity property.

This concludes the proof of Theorem 5.1. □

6. The Package Merge algorithm. Recall that our technique of constructing the optimal forest described in the Package Search algorithm is to compute all level-0 optimal packages. By the decomposition property (Theorem 5.1, part 1), we know that an optimal level-ℓ package is a union of two optimal level-$(\ell + 1)$ items. More precisely, in order to find the optimal package, $p^{\ell}_{[1,n]}(r)$, in each iteration of the inner loop of the Package Search algorithm, we need to find the minimum weight pair of optimal level-$(\ell + 1)$ items over $[1, n]$ that are level-ℓ connected with respect to $F[1, n, n - r + 2, \ell]$. This restricts our search domain to $O(n^2)$ pairs.

In this section we show that we can further restrict the search domain by considering only pairs of optimal items that are "tentatively connected," a notion first introduced by Hu and Tucker [8], and generalized in this paper.

(Since, by Theorem 5.1, we need only consider optimal items over the interval $[1, n]$, we refer to optimal items over the interval $[1, n]$ simply as "optimal items" throughout this section.)

DEFINITION. *Suppose that Q is a skyline. We say that two sets of tiles, p_1, p_2 are* tentatively connected with respect to Q *if Q has no gaps in the interval*

$$[min(lboundary\ (p_1), lboundary\ (p_2)) + 1, max(lboundary\ (p_1), lboundary\ (p_2)) - 1].$$

The main result of this section is stated in the following theorem.

THEOREM 6.1. *Let F be a level-ℓ optimal forest over $[1, n]$ and p_1, p_2 be the smallest total weight pair of optimal level-$(\ell + 1)$ items that are tentatively connected with respect to F. Then $p_1 \cup p_2$ is the minimal weight level-ℓ glue-pack with respect to F.*

First, we examine the properties of the minimum weight pair of level-$(\ell + 1)$ optimal items that are tentatively connected with respect to a level-ℓ optimal forest.

LEMMA 6.2. *Let Q be a level-ℓ skyline over $[1, n]$ that is a union of B^ℓ and level-$(\ell + 1)$ optimal items. Let $p \not\subseteq Q$ be an optimal level-$(\ell + 1)$ package that is not a glue-pack with respect to Q. Then there exists an optimal level-$(\ell + 1)$ item $q \not\subseteq Q$, such that $|q| < |p|$ and p, q are tentatively connected with respect to Q.*

Proof. Assume first that p spans a gap of Q. Let $r_1, r_2, \ldots, r_k$, where $r_t < r_{t+1}$, be the gaps of Q spanned by p. Then p and $s_{r_1,\ell}$ are tentatively connected with respect to Q. By the package span property (Theorem 5.1, part 3), $|p| > |s_{r_1,\ell}|$.

Suppose p does not span a gap of Q. Let $[i, j] = [lboundary\ (p), rboundary(p)]$. Let $F = F[1, n, n - t + 1, \ell + 1]$, where t is that index for which $p = p^{\ell+1}_{[1,n]}(t)$. By the definition of optimal package, p is a level-$(\ell + 1)$ glue-pack with respect to F. By Lemma 3.5, p is a glue-pack with respect to $F \cup Q$. By the monotonicity property (Theorem 5.1, part 2), F is the union of $B^{\ell+1}$ and all optimal level-$(\ell + 1)$ packages of weight less than $|p|$. We next show that there is some optimal level-$(\ell + 1)$ package $q' \not\subseteq Q$ such that $|q'| < |p|$ and q' has some tiles in the interval $[i, j]$. Suppose there is no such q. Then, since p spans no gap of F, this implies that $(F \cup Q)_{[i,j]} = Q_{[i,j]}$. Applying Lemma 3.7 twice, we see that p is a level-$(\ell + 1)$ glue-pack with respect to Q, contradicting our hypothesis.

We consider two cases: when q' spans a gap of Q, and when it does not.

Case 1. If q' does not span a gap of Q, then $p \cup q'$ also does not span a gap, and therefore p and q' are tentatively connected. This implies the lemma with $q = q'$.

Case 2. Assume that q' spans at least one gap of Q. Let $r_1, r_2, \ldots, r_k$, where $r_t < r_{t+1}$, be the gaps of Q spanned by q'. If $r_1 \leq i$, let r be the greatest r_t such that $r_t \leq i$, and then Q has no gaps in the interval $[r + 1, i - 1]$. Otherwise, let $r = r_1$, and then Q has no gaps in the interval $[i + 1, r - 1]$.

Thus, p and $s_{r,\ell+1}$ are tentatively connected with respect to Q. By the Package Span property, $|s_{r,\ell+1}| < |q'|$, thus $|s_{r,\ell+1}| < |p|$. This implies the lemma with $q = s_{r,\ell+1}$.

This concludes the proof of Lemma 6.2. □

Proof (of Theorem 6.1). Let p be the minimal weight level-ℓ glue-pack with respect to F. By Theorem 5.1, $p = p_1 \cup p_2$, where p_1, p_2 are optimal level-$(\ell + 1)$ items which must be tentatively connected, since otherwise $F \cup p$ would not be level-ℓ connected.

Let p_1, p_2 be the smallest pair of optimal level-$(\ell + 1)$ items that are tentatively connected with respect to F. Let $|p_1| < |p_2|$. By Lemma 6.2, p_1 is a glue-pack with respect to F (otherwise p_1, p_2 would not be the minimal pair).

We need to prove that p_2 is a level-$(\ell + 1)$ glue-pack with respect $F \cup p_1$. If p_2 is a tile, we are done. We can thus assume that p_2 is a package. Suppose p_2 is not a level-$(\ell + 1)$ glue-pack with respect to $F \cup p_1$. If $F \cup p_1$ has a gap at $i_2 = lboundary\ (p_2)$, then $s_{i_2,\ell+1}$, p_1 form a tentatively connected pair of total weight smaller than the pair p_1, p_2, contradicting the minimality of that pair. Otherwise, by Lemma 6.2, choose an optimal level-$(\ell + 1)$ item, $q \not\subseteq F \cup p_1$, such that $|q| < |p_2|$ and q and p_2 are tentatively connected with respect to $F \cup p_1$.

It is a routine matter of checking cases to verify that p_1 and q are tentatively connected, contradicting the minimality of the pair p_1, p_2. □

We now describe the Package Merge implementation of the Package Search algorithm introduced in §4.3. Each iteration of the main loop makes use of a data structure (the details of which are described below) which holds all optimal items (tiles, as well as packages produced during the previous iteration) and which allows rapid retrieval of the minimal tentatively connected pair.

During each iteration of the inner loop, the minimal tentatively connected pair of optimal level-$(\ell + 1)$ items is deleted from the data structure, and their union forms an optimal level-ℓ package.

Correctness of the implementation follows immediately from Theorem 6.1.

ALGORITHM 2. **Package Merge (L)**

```
1. for $\ell = L - 1$ downto 0 do
    1.1. $F^\ell \leftarrow B^\ell$;
    1.2. Initialize the data structure to be all optimal level-$(\ell + 1)$ items;
    1.3. for $r := 1$ to $n - \lceil n2^{\ell - L} \rceil - 1$ do
        1.3.1. $p_1, p_2 \leftarrow$ the minimal weight pair of tentatively connected
               optimal items;
        1.3.2. $p^\ell_{[1,n]}(r) \leftarrow p_1 \cup p_2$;
        1.3.3. $F^\ell := F^\ell \cup p^\ell_{[1,n]}(r)$;
        1.3.4. Delete $p_1$ and $p_2$ from the data structure;
2. compute the (standard) tree representation of $F^0$.
```

6.1. The data structure. The data structure consists of *local* priority queues C_0, $C_1, \ldots C_n$, where each C_i contains optimal level-$(\ell + 1)$ items prioritized by weight, together with a *global* priority queue G containing pairs of optimal level-$(\ell + 1)$ items, prioritized by total weight.

We refer to an item (that is, an optimal level-$(\ell + 1)$ item) as *unused* if it is still in the data structure, and *used* if it is not, i.e., it has already been chosen as one of the two halves of an optimal level-ℓ package.

At any given time, every unused tile is a member of exactly two of the local priority queues, while every unused package is a member of exactly one of the local priority queues.

The data structure is initialized by letting C_0 consist only of $s_{1,\ell+1}$, C_n consist only of $s_{n,\ell+1}$, and, for $i \in [1, n - 1]$, C_i consist of $s_{i,\ell+1}$, $s_{i+1,\ell+1}$, and all optimal level-$(\ell + 1)$ packages whose *lboundary* is equal to i. G will, at all times, contain the minimal pair of items from each local priority queue that has at least two items. Initially, G has $(n - 1)$ members, namely the pairs $\{s_{i,\ell+1}, s_{i+1,\ell+1}\}$ for all $i \in [1, n - 1]$.

At any time, any two unused items will be tentatively connected if and only if they are members of the same local priority queue. Thus, the minimal tentatively connected pair will always be the minimal item of G.

Deletion of the minimal pair of items is implemented as follows. Each of the two items is deleted from each of the local priority queues that it belongs to: one if a package, two if a tile. Deletion of a tile from two local priority queues causes the two queues to be merged. For each of the one, two, or three old local priority queues which have been modified, minimal pairs must be deleted from G (not necessarily the "deletemin" operation) and a new minimal pair for the new local priority queue is computed and inserted into G.

Time Complexity. Initializing the data structure requires $O(n)$ time. Each deletion from the data structure takes $O(\log n)$ time. Thus, the overall time complexity of the algorithm is $O(nL \log n)$.

7. Appendix: Resolving ties.

At some step during the execution of the Package Merge algorithm, there may be more than one choice of tentatively connected pairs that have minimum total weight. We proved correctness of the algorithm under the assumption that this never occurs—in fact, we made the even stronger assumption that the uniqueness hypothesis (introduced in §4) holds. In this section we show that this assumption is unnecessary.

In practice, the Package Merge algorithm works by resolving all ties according to specific rules given in this section. The correctness of this method is proved by showing that it is equivalent to increasing the weight of each tile by an "infinitesimal" amount, where the "infinitesimals" are chosen so as to guarantee the uniqueness hypothesis, without violating the monotonicity condition on tile weights.

7.1. Resolve a tie by choosing the smallest tag.

We assign each tile a unique positive integral *tag* value, taking care that the tags of tiles of the same index are monotone in the level. For example, the tile $s_{i,\ell}$ could be assigned the tag value $n\ell + i$. Each nonempty set of tiles will be assigned the largest tag value of any of its member tiles.

If at any time during execution of the Package Search algorithm, or its implementation, the Package Merge algorithm, there is a choice among two sets of tiles of the same actual weight, the set with the smaller tag value will be treated as the one of smaller weight.

This technique requires a slight clarification, since the algorithm requires searching for the least-weight set of tiles which has a certain property, and it could be that two or more such sets have the same tag value, as they could have tiles in common. If p and q are sets of tiles which have the same weight, and if the algorithm requires choosing the one of smaller weight, we choose p if and only if the tag value of $p - q$ is less than the tag value of $q - p$. These cannot be equal, since $p - q$ and $q - p$ must be disjoint.

Keeping track of tag values for all the sets concerned does not increase the asymptotic time complexity of the Package Merge algorithm, since it only requires storing one additional integer with each optimal item, and using this tag to break ties in the priority queues. Within each priority queue, the least-weight pair is still obtained by combining the two items of smallest weight, where "smallest" is now unambiguous, using the tie-breaker. Finding that priority queue which has the smallest least-weight pair is also easy using the tie-breaker.

7.2. Infinitesimal penalties.

Let $\epsilon > 0$ be a real number so small that, if A, B are any sets of tiles such that $|A| < |B|$, then $|A| + \epsilon < |B|$. (Since there are finitely many sets of tiles, such an ϵ must exist.) For each tile s, let $t(s)$ be the tag value of s. Let T be an integer larger than the largest tag value of any tile, and let each tile s be assigned the *fictitious* weight $|s| + \epsilon \cdot 2^{t(s)-T}$. If A and B are sets of tiles, we say that $A \prec B$ if the fictitious weight of A is less than the fictitious weight of B.

Remark 7.1. If A and B are sets of tiles, then we have the following properties.

1. Uniqueness: If $A \neq B$, then $A \prec B$ or $B \prec A$.
2. Consistency: If $|A| < |B|$, then $A \prec B$.
3. Same as tag scheme: If $|A| = |B|$, then the tie-breaking scheme of §7.1 will pick A to be the "lesser weight" if and only if $A \prec B$.
4. Nonnegativity: The fictitious weight of any set of tiles is nonnegative.
5. Monotonicity: For any $1 \leq i \leq n$ and any $0 \leq \ell < L$, $s_{i,\ell} \prec s_{i,\ell+1}$.

Thus, using fictitious weights instead of actual weights causes the uniqueness hypothesis to hold, without losing the needed conditions of nonnegativity and monotonicity introduced in §4, and the tie-breaking scheme always chooses that set which has the lighter fictitious weight.

Furthermore, at the end of the algorithm, when the optimal level-0 tree (i.e., tree of least fictitious weight) over $[1, n]$ is found, we are guaranteed that this geometric tree has the least actual weight, and hence represents an optimal solution to the original weighted binary tree problem.

8. Appendix: Proof of the interleaving property. In this section we give a proof of Theorem 4.3. The proof is very similar to proofs given in [1]–[4], [10]–[12], [16], and [17].

LEMMA 8.1 (Monge property of optimal forests). *If* $1 \le M \le M' \le N \le N' \le n$, $0 \le \ell \le L$, *and* $(N' - M + 1)2^{\ell - L} \le k \le (N - M' + 1)$, *then*

$$|F[M, N, k, \ell]| + |F[M', N', k, \ell]| \le |F[M, N', k, \ell]| + |F[M', N, k, \ell]| .$$

Proof. The proof is by backwards induction on ℓ, and for given ℓ, by induction on k.

We first note that if $M = M'$ or $N = N'$, both sides of the inequality are the same, so the result is trivial. Thus, without loss of generality, $M < M'$ and $N < N'$.

If $\ell = L$ the result is vacuous, since no k satisfies the condition.

Suppose $k > 1$. Let $c = c_{k-1}(F[M', N, k, \ell])$ and $c' = c_{k-1}(F[M, N', k, \ell])$. We consider two cases: $c \le c'$ and $c > c'$. If $c \le c'$, the inductive hypothesis gives us

$$|F[M, c, k-1, \ell]| + |F[M', c', k-1, \ell]| \le |F[M, c', k-1, \ell]| + |F[M', c, k-1, \ell]| .$$

Thus

$$\begin{aligned} &|F[M, N, k, \ell]| + |F[M', N', k, \ell]| \\ &\le |F[M, c, k-1, \ell]| + |F[c+1, N, 1, \ell]| + |F[M', c', k-1, \ell]| + |F[c'+1, N', 1, \ell]| \\ &\le |F[M, c', k-1, \ell]| + |F[c'+1, N', 1, \ell]| + |F[M', c, k-1, \ell]| + |F[c+1, N, 1, \ell]| \\ &= |F[M, N', k, \ell]| + |F[M', N, k, \ell]| . \end{aligned}$$

If $c \ge c'$, the inductive hypothesis gives us

$$|F[c'+1, N, 1, \ell]| + |F[c+1, N', 1, \ell]| \le |F[c'+1, N', 1, \ell]| + |F[c+1, N, 1, \ell]| .$$

Thus

$$\begin{aligned} &|F[M, N, k, \ell]| + |F[M', N', k, \ell]| \\ &\le |F[M, c', k-1, \ell]| + |F[c', N, 1, \ell]| + |F[M', c, k-1, \ell]| + |F[c+1, N', 1, \ell]| \\ &\le |F[M, c', k-1, \ell]| + |F[c'+1, N', 1, \ell]| + |F[M', c, k-1, \ell]| + |F[c+1, N, 1, \ell]| \\ &= |F[M, N', k, \ell]| + |F[M', N, k, \ell]| . \end{aligned}$$

Finally, suppose $k = 1$, $\ell < L$. We consider two cases: $M' < N$, and $M' = N$. If $M' < N$, then, by the inductive hypothesis,

$$|F[M, N, 2, \ell+1]| + |F[M', N', 2, \ell+1]| \le |F[M, N', 2, \ell+1]| + |F[M', N, 2, \ell+1]| .$$

Let $c = c_1(F[M', N, 2, \ell+1])$, and let $c' = c_1(F[M, N', 2, \ell+1])$. By symmetry, we can assume without loss of generality that $c \le c'$. Thus, since

$$|B^{\ell}_{[M,N]}| + |B^{\ell}_{[M',N']}| = |B^{\ell}_{[M,N']}| + |B^{\ell}_{[M',N]}|$$

we have

$$\begin{aligned} &|F[M, N, 1, \ell]| + |F[M', N', 1, \ell]| \\ &= |F[M, N, 2, \ell+1]| + |B^{\ell}_{[M,N]}| + |F[M', N', 2, \ell+1]| + |B^{\ell}_{[M',N']}| \\ &\le |F[M, N', 2, \ell+1]| + |B^{\ell}_{[M,N']}| + |F[M', N, 2, \ell+1]| + |B^{\ell}_{[M',N]}| \\ &= |F[M, N', 1, \ell]| + |F[M', N, 1, \ell]| . \end{aligned}$$

In the case $M' = N$, we need an additional lemma.

LEMMA 8.2. *If* $L - \ell - 1 \geq \log_2 n$ *then*

$$|F[M, N, 1, \ell]| + |s_{N,\ell+1}| \leq \left|F[M, N, 1, \ell + 1] \cup B^{\ell}_{[M,N]}\right|.$$

Proof. Let ℓ_i be the greatest level of any tile of index i in $F[M, N, 1, \ell]$. Then

$$\left|F[M, N, 1, \ell + 1] \cup B^{\ell}_{[M,N]}\right| - |F[M, N, 1, \ell]| \geq \sum_{i=M}^{N} |s_{i,\ell_i+1}|$$

which is at least $|s_{N,\ell+1}|$. □

We return to the proof of Lemma 8.1. Let $c = c_1(M, N', 2, \ell + 1)$. We assume that $c \geq N$. (The case where $c < N$ is symmetric.) By the inductive hypothesis,

$$|F[M, N, 1, \ell + 1]| + |F[N, c, 1, \ell + 1]| \leq |s_{N,\ell+1}| + |F[M, c, 1, \ell + 1]|.$$

Then, by Lemma 8.2, we have

$$\begin{aligned}
&|F[M, N, 1, \ell]| + |F[N, N', 1, \ell]| \\
&\leq |F[M, N, 1, \ell]| + |F[N, c, 1, \ell + 1]| + |F[c + 1, N', 1, \ell + 1]| + \left|B^{\ell}_{[N,N']}\right| \\
&\leq \left|B^{\ell}_{[M,N]}\right| + |F[M, N, 1, \ell + 1]| - |s_{N,\ell+1}| \\
&+ |F[N, c, 1, \ell + 1]| + |F[c + 1, N', 1, \ell + 1]| + \left|B^{\ell}_{[N,N']}\right| \\
&\leq |F[M, c, 1, \ell + 1]| + \left|B^{\ell}_{[M,N]}\right| + |F[c + 1, N', 1, \ell + 1]| + \left|B^{\ell}_{[N,N']}\right| \\
&= |F[M, N', 1, \ell]| + |s_{N,\ell}|.
\end{aligned}$$

This concludes the proof of Lemma 8.1. □

We are now ready to give the proof of the interleaving property.

LEMMA 8.3 (interleaving property). *For* M, N, k, ℓ, *if* $1 \leq M < N \leq n$, $0 \leq \ell \leq L$, $(N - M + 1)2^{\ell-L} \leq k \leq N - M$, *and* $0 \leq m \leq k$, *then*

$$c_m(F[M, N, k + 1, \ell]) \leq c_m(F[M, N, k, \ell]) \leq c_{m+1}(F[M, N, k + 1, \ell]).$$

Proof. It suffices to show that $c_m(F[M, N, k, \ell]) \leq c_{m+1}(F[M, N, k + 1, \ell])$. The other inequality is symmetric.

Let $c = c_{m+1}(F[M, N, k + 1, \ell])$ and $c' = c_m(F[M, N, k, \ell])$. Suppose $c' < c$.

Let $M' = c_1(F[M, N, k + 1, \ell])$. By Lemma 4.2,

$$F[M', N, k, \ell] = F[M, N, k + 1, \ell]_{[M',N]}.$$

Thus, $c = c_m(F[M', N, k, \ell])$.

By the uniqueness hypothesis,

$$\begin{aligned}
|F[M, c', m, \ell]| + |F[c' + 1, N, k - m, \ell]| &< |F[M, c, m, \ell]| + |F[c + 1, N, k - m, \ell]|, \\
|F[M', c, m, \ell]| + |F[c + 1, N, k - m, \ell]| &< |F[M', c', m, \ell]| + |F[c' + 1, N, k - m, \ell]|.
\end{aligned}$$

Adding the two inequalities and cancelling, we obtain

$$|F[M, c', m, \ell]| + |F[M', c, m, \ell]| < |F[M, c, m, \ell]| + |F[M', c', m, \ell]|,$$

which contradicts Lemma 8.1. This concludes the proof of the interleaving property. □

Acknowledgment. We wish to thank the referee for devoting the very substantial amount of time that was necessary to review this paper, and for giving us so many valuable suggestions.

REFERENCES

[1] A. APOSTOLICO M. J. ATALLAH, L. L. LARMORE, AND H. S. MCFADDIN, *Efficient parallel algorithms for string editing and related problems*, Proc. 26th Allerton Conference on Communication, Control, and Computing, Monticello, IL 1988, pp. 253–263. Reprinted as CSD–TR-724, Purdue University, West Lafayette, IN, 1988, reprinted in SIAM J. Comput., 19 (1990), pp. 968–988.

[2] W. W. BEIN, P. BRUCKER, J. K. PARK, AND P. K. PATHAK, *A Monge property for the d-dimensional transportation problem*, Technical Report CS92-1, Department of Computer Science, University of New Mexico, Albuquerque, NM, 1992.

[3] W. W. BEIN AND P. K. PATHAK, *A characterization of the Monge property*, Technical Report CS90-10, Department of Computer Science, University of New Mexico, Albuquerque, NM, (1990).

[4] M. R. GAREY, *Optimal binary search trees with restricted maximal depth*, SIAM J. Comput., 3 (1974), pp. 101–110.

[5] A. M. GARSIA AND M. L. WACHS, *A new algorithm for minimal binary search trees*, SIAM J. Comput., 6 (1977), pp. 622–642.

[6] E. N. GILBERT, *Codes based on inaccurate source probabilities*, IEEE Trans. Inform. Theory, 17 (1971), pp. 304–314.

[7] T. C. HU, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.

[8] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable length alphabetic codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[9] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proceedings of the Institute of Radio Engineers, 40 (1952), pp. 1098–1101.

[10] A. ITAI, *Optimal alphabetic trees*, SIAM J. Comput., 5 (1976), pp. 9–18.

[11] D. E. KNUTH, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14–25.

[12] L. L. LARMORE, *Height restricted optimal alphabetic trees*, SIAM J. Comput., 16 (1987), pp. 1115–1123.

[13] ———, *Length limited coding and optimal height-limited binary trees*, Tech. Rep. 88-01, ICS Dept., University of California, Irvine, CA, 1988.

[14] L. L. LARMORE, AND D. S. HIRSCHBERG, *A fast algorithm for optimal length-limited Huffman codes*, J. Assoc. Comput. Mach., 37 (1990), pp. 464–473.

[15] ———, *Length-limited coding*, Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, January, 1990, pp. 310–318.

[16] R. L. WESSNER, *Optimal alphabetic search trees with restricted maximal height*, Inform. Process. Lett., 4 (1976), pp. 90–94.

[17] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, Proceedings 12th ACM Symposium on Theory of Computing, (1980), pp. 429–435.

[18] Y. ZHU AND J. WANG, *On Alphabetic-extended binary trees with restricted path length*, Scientia Sinica, 22 (1979), pp. 1362–1371.

SIAM J. COMPUT.
Vol. 23, No. 6, pp. 1313–1347, December 1994

013

IMPROVED ALGORITHMS FOR LINEAR INEQUALITIES WITH TWO VARIABLES PER INEQUALITY*

EDITH COHEN† AND NIMROD MEGIDDO‡

Abstract. The authors show that a system of m linear inequalities with n variables, where each inequality involves at most two variables, can be solved in $\tilde{O}(mn^2)$ time (we denote $\tilde{O}(f) = O(f \operatorname{polylog} n \operatorname{polylog} m)$) deterministically, and in $\tilde{O}(n^3 + mn)$ expected time using randomization. Parallel implementations of these algorithms run in $\tilde{O}(n)$ time, where the deterministic algorithm uses $\tilde{O}(mn)$ processors and the randomized algorithm uses $\tilde{O}(n^2 + m)$ processors. The bounds significantly improve over previous algorithms. The randomized algorithm is based on novel insights into the structure of the problem.

Key words. linear inequalities, two-variables inequalities, linear systems, linear programming, strongly polynomial algorithms

AMS subject classifications. 90C27, 05C85, 90C05

1. Introduction.

In this paper we consider the following class of linear systems.

DEFINITION 1.1. *A* TVPI *system is a system of linear inequalities where each inequality involves at most two variables. (I.e., a system of the form* $\boldsymbol{Ax} \le \boldsymbol{b}$ *, where* $\boldsymbol{A} \in R^{m\times n}$ *is a matrix,* $\boldsymbol{b} \in R^m$ *is a real vector, and each row of* $\boldsymbol{A}$ *contains at most two nonzero entries.) We denote the number of inequalities by m, and the variables by* $x_1, \ldots, x_n$*. We denote by* $F = \{\boldsymbol{x} \in R^n | \boldsymbol{Ax} \le \boldsymbol{b}\}$ *the set of feasible solutions. A* TVPI *system is called* monotone *if the two nonzero entries in each row have opposite signs. See Fig.* 1 *for an example of a* TVPI *system.*

Our goal is to either find a point that satisfies all the inequalities or conclude that no such point exists. The structure of TVPI systems enables us to obtain specialized algorithms that are faster than known algorithms for solving general linear systems. The algorithms given here can also be adapted to find a solution that minimizes or maximizes a specific variable. We summarize previous work on solving TVPI systems. Shostak [17] characterized the set of solutions of TVPI systems and gave an algorithm that is exponential in the worst case. Nelson [16] gave an $n^{O(\log n)}$ algorithm. Aspvall and Shiloach [3] and Aspvall [2] proposed algorithms that perform $O(mn^3 I)$ and $O(mn^2 I)$ arithmetic operations, respectively, where I is the size of the binary encoding of the problem. Megiddo [15] proposed the first strongly polynomial time algorithm for the problem, which performs $O(mn^3 \log m)$ operations. The parallel implementation of Megiddo's algorithm runs in $O(n^3 \log m)$ time using $O(m)$ processors.

The algorithms presented here improve the sequential and parallel time bounds. We give an $O\left(mn^2(\log m + \log^2 n)\right)$ time deterministic algorithm, which has a parallel implementation that runs in $O\left(n(\log m + \log^2 n)\right)$ time using $O(mn)$ processors. An additional improvement is obtained through using randomization: We give an algorithm that runs in $O\left(n^3 \log n + mn(\log^5 n + \log m \log^3 n)\right)$ expected time. A parallel implementation runs in $O\left(n(\log^5 n + \log^3 n \log m)\right)$ expected time using $O(n^2 + m)$ processors. The space requirement of the algorithms presented here is $O(n^2 + m)$. The effort involved in translating these

*Received by the editors July 28, 1991; accepted for publication September 20, 1993.

†AT&T Bell Labs, Murray Hill, New Jersey 07974 (edith@research.att.com). Research done while the author was at Stanford University and IBM Almaden Research Center. The research of this author was partially supported by Office of Naval Research grant N00014-91-C-0026 and by National Science Foundation PYI grant CCR-8858097, matching funds from AT&T, and Digital Equipment Corporation.

‡IBM Research, Almaden Research Center, San Jose, California 95120-6099 and School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel (megiddo@almaden.ibm.com). The research of this author was partially supported by Office of Naval Research grant N00014-91-C-0026.

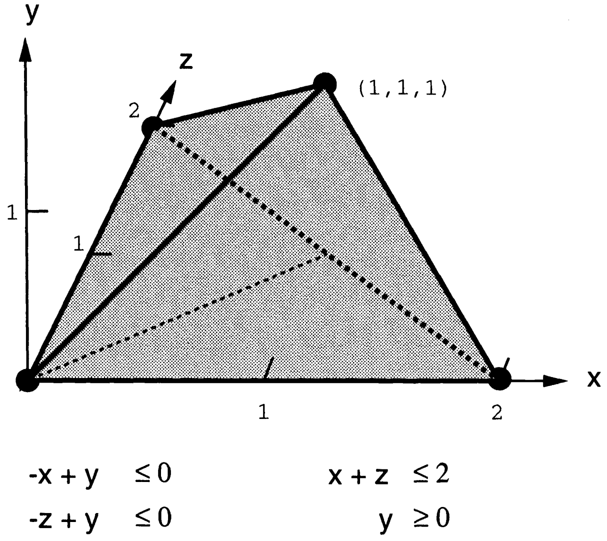


FIG. 1. *An example of a* TVPI *system with three variables.*

algorithms into actual programs is about the same as in the previously known algorithms, and the hidden constants in the time bounds are still reasonable. Subsequent to our work, Hochbaum and Naor [12] found a new $O(mn^2 \log m)$ deterministic algorithm for the problem. Their algorithm, however, runs in $O(n^2 \log m)$ time in parallel, and it does not seem possible to combine it with the randomized approach to yield algorithms with better expected time.

Section 2 gives the preliminaries. We give a characterization of polyhedra defined by TVPI systems. We represent a TVPI system by a directed multigraph with $2n$ vertices and $2m$ edges. The edges have linear functions of a single variable as "weights," and the weight of a path is interpreted as compositions of the weights along the path's edges. We review a result of Shostak [17] that relates weights of directed cycles and paths in this multigraph to bounds on the feasible region. We also discuss some subproblems and show how to solve them by performing a "Bellman-Ford shortest-path"-type computation on the representing multigraph where initial numerical values are assigned to vertices and "adding" an edge weight is replaced by substituting the current value at the vertex in the linear function comprising the edge weight (hence by an $O(mn)$ sequential algorithm, or by an $O(n)$ time parallel algorithm using $O(m)$ processors). In particular we discuss the subproblem of *locating a value*: For a numerical value ξ and $1 \leq i \leq n$, decide whether a TVPI system with the addition of a constraint of the form $x_i \geq \xi$ or $x_i \leq \xi$ remains feasible. An $O(mn)$ algorithm for locating a value was given by Aspvall and Shiloach [3].

In §3 we introduce a framework for solving TVPI systems that amounts to performing a "Floyd-Warshall all-pairs shortest-path"-type computation where resolving comparisons is replaced by locating values. This formulation enables us to reduce solving TVPI systems to performing $\tilde{O}(m + n^3)$ operations for the Floyd-Warshall-type computation and in addition solving a polylogarithmic number of instances of the more general subproblem of *locating a pool of values*: For numerical values $\xi_1, \ldots, \xi_n$ choose bounds $s_1, \ldots, s_n$, where $s_i \in \{x_i \geq \xi_i, x_i \leq \xi_i\}$, such that the system with the additional constraints $s_1, \ldots, s_n$ remains feasible. Obviously, a pool of values can be located by n sequential applications of an algorithm that locates a single value. This yields an $O(mn^2)$ time sequential algorithm that in parallel, runs in $O(n^2)$ time using $O(m)$ processors. The latter yields a $\tilde{O}(mn^2)$ deterministic algorithm for solving TVPI systems.

In §4 we present better algorithms for locating a pool. We first give an algorithm that improves the parallel running time to $O(n)$ using $O(mn)$ processors. This algorithm enables us to obtain a $\tilde{O}(n)$ parallel deterministic running time (using $O(mn)$ processors) for solving TVPI systems. Section 4 also contains an overview of a two stage $\tilde{O}(mn)$ time randomized algorithm. In parallel, the randomized algorithm runs in $\tilde{O}(n)$ time and uses $O(m)$ processors. This algorithm enables us to obtain the stated randomized bounds for solving TVPI systems. The randomized algorithm constitutes the most novel and the most technically difficult part of this work. Sections 5 and 6 are concerned with the details of the two stages.

In §7 we discuss the special structure of monotone systems. These systems are of special interest since the linear programming duals of generalized transshipment problems are monotone. In [4] and [7] the authors exploited this relationship and presented algorithms for generalized network flow problems (see also Adler and Cosares [1]).

Section 8 contains concluding remarks and a discussion of open problems.

2. Preliminaries.

Consider two two-variable inequalities that share a common variable. In some cases, these inequalities can be combined to yield a new two-variable inequality that is implied by the previous two. For example, the inequalities $x_1 \leq 2x_2 - 4$ and $x_2 \leq 4x_3 + 5$ can be combined to yield $x_1 \leq 8x_3 + 6$. The inequalities $x_1 \leq 2x_2 - 4$ and $x_2 \geq 4x_3 + 5$, however, cannot be combined in a similar fashion. The notion of combining inequalities was first discussed by Shostak [17] and used in all succeeding works. A TVPI system was represented by a multigraph where nodes correspond to variables and inequalities correspond to edges. Each set of inequalities that can be combined as described above to yield a new inequality forms an *admissible* path in this multigraph [17]. We find it more convenient to consider a slightly different graph (the *associated graph* of a system), in which there is an exact correspondence between directed paths and new inequalities that can be obtained as above. Cycles in the associated graph yield bounds on variables. Obviously, the new inequalities and bounds that are obtained by combining inequalities are implied by the original system. Shostak proved that the converse is also true: The tightest bounds on variables that can be obtained from simple cycles and paths define the bounding box of the feasible region. The current paper, like much previous work (see [3], [2], [15], and [17]), makes use of Shostak's result.

2.1. The associated graph.

We represent a TVPI system by a set of n intervals and a directed multigraph with $2n$ nodes and $2m$ edges. The graph is a natural representation of the system, where variables correspond to vertices and two-variable inequalities to edges between vertices of the participating variables. A formal definition of the associated graph follows.

DEFINITION 2.1. *Suppose we are given a* TVPI *system as in Definition* 1.1. *Without loss of generality, assume that the inequalities with a single variable* (bounds) *are summarized in the form of intervals* $S_i = [a_i, b_i]$ $(-\infty \leq a_i \leq b_i \leq \infty,\ i = 1, \dots, n)$. *We consider a directed multigraph* $G = (V, E)$ *as follows. For each variable* x_i, *there are two vertices in* G *associated with* x_i, *namely,* $V = \overline{V} \cup \underline{V}$ *where* $\overline{V} = \{\overline{v}_i \mid i = 1, \dots, n\}$ *and* $\underline{V} = \{\underline{v}_i \mid i = 1, \dots, n\}$. *For* $u \in V$, *we define* $u^{-1} \in V$ *as follows. If* $u = \underline{v}_i \in \underline{V}$, *then* $u^{-1} = \overline{v}_i$, *and if* $u = \overline{v}_i \in \overline{V}$, *then* $u^{-1} = \underline{v}_i$. *The edges of* G *are associated with the inequalities that involve exactly two variables. Note that each such inequality can be written in the form*

$$\gamma x_i \leq \alpha x_j + \beta,$$

where either $\gamma = 1$ *and* $\alpha \neq 0$ *or* $\gamma = -1$ *and* $\alpha > 0$. *Each such inequality is represented by two edges, where each edge* e *is labeled with a certain linear function* f_e *as follows*:

1. *If* $\alpha > 0$ *and* $\gamma = 1$, *we have an edge* $e = (\overline{v}_j, \overline{v}_i)$ *labeled* $f_e(x) = \alpha x + \beta$ *and an edge* $e^{-1} = (\underline{v}_i, \underline{v}_j)$ *labeled* $f_{e^{-1}}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.
2. *If* $\alpha < 0$ *and* $\gamma = 1$, *we have an edge* $e = (\underline{v}_j, \overline{v}_i)$ *labeled* $f_e(x) = \alpha x + \beta$ *and an edge* $e^{-1} = (\underline{v}_i, \overline{v}_j)$ *labeled* $f_{e^{-1}}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$.
3. *If* $\alpha > 0$ *and* $\gamma = -1$, *we have an edge* $e = (\overline{v}_j, \underline{v}_i)$ *labeled* $f_e(x) = -\alpha x - \beta$ *and an edge* $e^{-1} = (\overline{v}_i, \underline{v}_j)$ *labeled* $f_{e^{-1}}(x) = -\frac{1}{\alpha}x - \frac{\beta}{\alpha}$.

See Fig. 2 *for an example of such a system and the associated graph. See Fig.* 3 *for the associated graph of the system shown in Fig.* 1. *We assume throughout this paper that a* TVPI *system is given by the associated graph and set of intervals.*

$$\begin{array}{llll}
(1.) & y & \le & x+2 \\
(2.) & y & \ge & x \\
(3.) & y & \ge & -x \\
(4.) & y & \le & -x+2
\end{array}$$

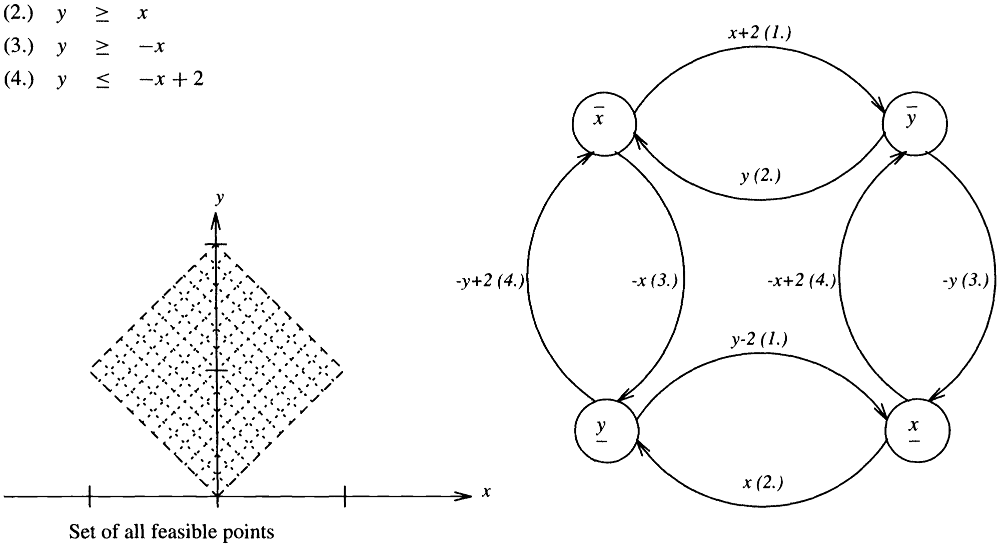


FIG. 2. *An example of a system of inequalities and the associated graph.*

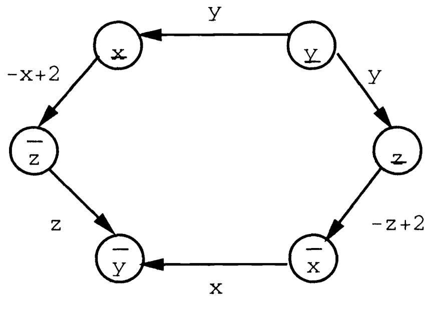


FIG. 3. *The associated graph of the* TVPI *system of Fig.* 1.

For any one-to-one function f let f^{-1} denote the inverse function. In particular, if $f(x) = \alpha x + \beta$ and $\alpha \ne 0$, then $f^{-1}(x) = \frac{1}{\alpha}x - \frac{\beta}{\alpha}$. Note that for all $e \in E$, $f_{e^{-1}} = f_e^{-1}$.

Let G be as in Definition 2.1. A linear function associated with an edge (u, w) corresponds to an inequality in the original system: Suppose that $u \in \{\underline{v}_i, \overline{v}_i\}$; if $w = \overline{v}_j$ the inequality is $x_j \le f_e(x_i)$, and if $w = \underline{v}_j$ the inequality is $x_j \ge f_e(x_i)$. We define the linear function associated with a directed path p from u to w. The corresponding two-variable inequality that results from treating the path as an edge (u, w), is implied by the original system.

DEFINITION 2.2. *Let $p = (e_1, \dots, e_k)$ be a (directed) path in G.*

1. *For any path p, we define a linear function f_p, where $f_p = f_{e_k} \circ \cdots \circ f_{e_1}$. Note that if both ends of p lie either in $\overline{V}$ or in $\underline{V}$ then f_p is increasing. Otherwise, f_p is decreasing.*
2. *We denote by p^{-1} the path $p = (e_k^{-1}, \dots, e_1^{-1})$. Note that $f_{p^{-1}} = f_p^{-1}$ and hence the two paths p, p^{-1} correspond to the same inequality.*

Consider, for example, the directed path from $\underline{y}$ to $\overline{z}$ in the graph of Fig. 3. The linear function associated with this path is $f = 2 - y$, and the corresponding inequality is $z \le 2 - y$.

In particular, Definition 2.2 applies to cycles (closed paths). Cycles play a special role since they give a relation that involves a single variable, from which a bound on this variable can be deduced. This is formalized in the following definition.

DEFINITION 2.3. *We refer to a path that starts and ends at vertices from $\{v, v^{-1}\}$ as a* cycle. *A path from v to v is a* closed cycle, *and to a path from v to v^{-1} is an* open cycle.

1. *Let c be a cycle starting at one of the vertices $\overline{v}_i$ and $\underline{v}_i$, and ending at a vertex $\overline{v}_i$ (resp., $\underline{v}_i$). Let f_c be the associated linear function. The bound on x_i implied by the cycle c follows from the inequality $x \le f_c(x)$ (resp., $x \ge f_c(x)$). Obviously, the implied inequality must hold for all feasible points. Note that if a cycle c starts at $\overline{v}_i$ and ends at $\underline{v}_i$ (resp., starts at $\underline{v}_i$ and ends at $\overline{v}_i$), then f_c is decreasing. Hence c implies a lower bound (resp., an upper bound) on x_i.*
2. *We say that the cycle c* contradicts *a value ξ_i of $\overline{v}_i$ (resp., $\underline{v}_i$) if the bound implied by c is of the form $x_i \ge \alpha$ (resp., $x_i \le \alpha$) whereas $\xi_i < \alpha$ (resp., $\xi_i > \alpha$).*

Consider, as an example, the graph of Fig. 2 and the cycle that consists of the path from $\underline{x}$ to $\overline{x}$. The function associated with this cycle is $f = -x + 2$ and the bound implied by the cycle is $x \le -x + 2$ (i.e., $x \le 1$).

2.2. Pushing bounds along edges. We maintain numerical values associated with the vertices of G: The value associated with $\overline{v}_i$ (resp., $\underline{v}_i$) is denoted by $\overline{x}_i$ (resp., $\underline{x}_i$). Intuitively, $\overline{x}_i$ and $\underline{x}_i$ correspond to upper and lower bounds, respectively, on the value of the variable x_i. We say that ξ is *tighter* than μ as a value of $\overline{x}_i$ (resp., $\underline{x}_i$), if $\xi < \mu$ (resp., $\xi > \mu$). We introduce the following operation that considers an edge $e = (u, v)$ and updates the bound associated with v, as implied by the inequality associated with e and the bound associated with u.

DEFINITION 2.4. *A* push *through an edge e is defined as the following operation:*

1. *If $e = (\overline{v}_j, \overline{v}_i)$, $\overline{x}_i \leftarrow \min\{\overline{x}_i, f_e(\overline{x}_j)\}$.*
2. *If $e = (\underline{v}_j, \underline{v}_i)$, $\underline{x}_i \leftarrow \max\{\underline{x}_i, f_e(\underline{x}_j)\}$.*
3. *If $e = (\overline{v}_j, \underline{v}_i)$, $\underline{x}_i \leftarrow \max\{\underline{x}_i, f_e(\overline{x}_j)\}$.*
4. *If $e = (\underline{v}_j, \overline{v}_i)$, $\overline{x}_i \leftarrow \min\{\overline{x}_i, f_e(\underline{x}_j)\}$.*

A push is said to be essential *if it actually modifies the value of $\underline{x}_i$ or $\overline{x}_i$.*

PROPOSITION 2.5. *Suppose $\underline{x}_i$ and $\overline{x}_i$ $(i = 1, \dots, n)$ are initialized to some values and X is the set of vectors $\boldsymbol{x} = (x_1, \dots, x_n)^T$ that* (i) *satisfy all the given inequalities and* (ii) *$\underline{x}_i \le x_i \le \overline{x}_i$ $(i = 1, \dots, n)$. The set X is invariant under pushes.*

Proof. Consider any $\boldsymbol{x} \in X$ and a push through an edge $e = (\overline{v}_j, \overline{v}_i)$. (The arguments for the other cases are similar.) Since $\boldsymbol{x}$ is a feasible solution we have $x_i \le f_e(x_j)$. Note that f_e is an increasing function when both ends of e lie either in $\overline{V}$ or in $\underline{V}$. Hence, from $x_j \le \overline{x}_j$, it follows that $x_i \le f_e(\overline{x}_j)$. Since $x_i \le \overline{x}_i$, we have $x_i \le \min\{\overline{x}_i, f_e(\overline{x}_j)\}$.

It is easy to see that a vector $\boldsymbol{x}$ solves a given TVPI system if and only if (i) $x_i \in S_i$ $(i = 1, \dots, n)$ and (ii) the set of values $\overline{x}_i = \underline{x}_i = x_i$ $(i = 1, \dots, n)$ is invariant under pushes.

We define a new operation that amounts to assigning simultaneously at all vertices the tightest values that could be obtained by applying pushes (see Definition 2.4).

DEFINITION 2.6. *Consider the graph G with some set of values at the vertices, $\underline{x}_i = \underline{x}_i^k$, $\overline{x}_i = \overline{x}_i^k$ $(i = 1, \ldots, n)$. A* push phase *on G is assigning, at the respective vertices, the set of values $\underline{x}_i^{k+1}$, $\overline{x}_i^{k+1}$ $(i = 1, \ldots, n)$ defined as follows:*

$$\overline{x}_i^{k+1} \leftarrow \min\left\{\overline{x}_i^k, \min_{e\in\text{in}\{\overline{v}_i\}}\left\{f_e(\overline{x}_j^k) \text{ if } e = (\overline{v}_j, \overline{v}_i), \text{ or } f_e(\underline{x}_j^k) \text{ if } e = (\underline{v}_j, \overline{v}_i)\right\}\right\},$$

$$\underline{x}_i^{k+1} \leftarrow \max\left\{\underline{x}_i^k, \max_{e\in\text{in}\{\underline{v}_i\}}\left\{f_e(\overline{x}_j^k) \text{ if } e = (\overline{v}_j, \underline{v}_i), \text{ or } f_e(\underline{x}_j^k) \text{ if } e = (\underline{v}_j, \underline{v}_i)\right\}\right\}.$$

Performing a *push phase* is computationally similar to a single phase of the Bellman-Ford shortest-path algorithm. It takes $O(m)$ time sequentially; $O(\log\log n)$ time using $O(m)$ processors or $O(1)$ expected time using $O(m)$ processors by a randomized algorithm on a CRCW PRAM; and $O(\log m)$ deterministic time using $O(m)$ processors on a CREW PRAM. (See [13].)

We use push phases as a basic operation in algorithms presented later. When repeated applications of push phases are performed, the initial set of values is denoted by $\underline{x}_i = \underline{x}_i^0$, $\overline{x}_i = \overline{x}_i^0$ $(i = 1, \ldots, n)$ and the respective values of $\underline{x}_i$ and $\overline{x}_i$ after the termination of the kth push phase are denoted by $\underline{x}_i^k$ and $\overline{x}_i^k$.

DEFINITION 2.7. *Suppose that a sequence of push phases is performed. Consider pairs consisting of a value and a vertex of the form $(\underline{x}_j^k, \underline{v}_j)$ or $(\overline{x}_j^k, \overline{v}_j)$ (where $k \geq 0$ and $1 \leq j \leq n$). The* predecessor *of a pair (ξ, v) is the pair (ξ', v') such that $e = (v', v)$ is the edge through which the essential push determined the value ξ, and $\xi' = f_e^{-1}(\xi)$ is the corresponding value at v'. If the predecessor is not uniquely defined we choose arbitrarily among the qualified pairs. A pair (ξ, v) does not have a predecessor if and only if ξ is the initial value at v.*

The essential path *of a pair (ξ, v) as above consists of the sequence of pairs $(\xi_1, v_1), \ldots, (\xi_\ell, v_\ell)$ and the corresponding sequence of edges $e_1, \ldots, e_{\ell-1}$ such that* (i) *(ξ_i, v_i) is the predecessor of (ξ_{i+1}, v_{i+1}) and e_i is the edge through which the essential push occurs $(i = 1, \ldots, \ell - 1)$,* (ii) *$(\xi_\ell, v_\ell) = (\xi, v)$, and* (iii) *$(\xi_1, v_1)$ does not have a predecessor.*

It is easy to see that the essential paths that correspond to $(\underline{x}_i^\ell, \underline{v}_i)$ and $(\overline{x}_i^\ell, \overline{v}_i)$ $(i = 1, \ldots, n)$ can be found within the same time bounds of performing ℓ push phases, simply by keeping track of all essential push operations.

We extend the notion of a push through an edge to directed paths. Let $p = (e_1, \ldots, e_k)$ be a (directed) path in G. A *push along p* is defined to be the composition of k successive pushes, through the edges $e_1, \ldots, e_k$. It is easy to see that for every $\ell \geq 0$ and a vertex v, after a sequence of ℓ push phases, the value at v is exactly the tightest between the initial value at v and the values obtained by pushing corresponding initial values at vertices along paths of size at most k that are going into v.

2.3. Properties of the feasible region. We discuss the relation between the bounds of the interval of feasible values of a variable and the tightest bounds implied by directed cycles. We first give the following definitions.

DEFINITION 2.8.

1. *For each variable x_i, let $[\underline{x}_i^*, \overline{x}_i^*]$ be the set of values in $[a_i, b_i]$ that are not contradicted by any simple cycle. Note that the two cycles c and c^{-1} imply the same bound. It follows that in order to find $\overline{x}_i^*$ (resp., $\underline{x}_i^*$), it suffices to consider cycles ending at $\overline{v}_i$ (resp., $\underline{v}_i$).*
2. *For $1 \leq i \leq n$ denote by $[x_i^{\min}, x_i^{\max}] \subseteq [\underline{x}_i^*, \overline{x}_i^*]$ the largest interval such that* (1) *for all $1 \leq j \leq n$ and for all simple paths p from a vertex $\overline{v}_j$ (resp., $\underline{v}_j$) to $\overline{v}_i$, we*

have $x_i^{\max} \leq f_p(\overline{x}_j^*)$ (*resp.*, $x_i^{\max} \leq f_p(\underline{x}_j^*)$) *and* (2) *for all* $1 \leq j \leq n$ *and for all simple paths* p *from a vertex* $\overline{v}_j$ (*resp.*, $\underline{v}_j$) *to* $\underline{v}_i$ *we have* $x_i^{\min} \geq f_p(\overline{x}_j^*)$ (*resp.*, $x_i^{\min} \geq f_p(\underline{x}_j^*)$).

3. *An interval* $I \subset R$ *is* infeasible *with respect to the variable* x_i *if* $I \cap [x_i^{\min}, x_i^{\max}] = \emptyset$. *A value* $\overline{x}_i$ (*resp.*, $\underline{x}_i$) *is* infeasible *if* $\overline{x}_i < x^{\min}$ (*resp.*, $\underline{x}_i > x^{\max}$).
4. *A value* $\overline{x}_i$ (*resp.*, $\underline{x}_i$) *is* consistent with an interval $[a, b]$ *if* $\overline{x}_i \geq b$ (*resp.*, $\underline{x}_i \leq a$). *A value* $\overline{x}_i$ (*resp.*, $\underline{x}_i$) *is* consistent *if* $\overline{x}_i \geq x_i^{\max}$ (*resp.*, $\underline{x}_i \leq x_i^{\min}$).

REMARK 2.9. *When the values* $\overline{x}_i^*$ *and* $\underline{x}_i^*$ $(i = 1, \ldots, n)$ *are given, we can compute* $x_i^{\min}$ *and* $x_i^{\max}$ *as follows. Initialize the values at the vertices to be* $\overline{x}_i^0 = \overline{x}_i^*$ *and* $\underline{x}_i^0 = \underline{x}_i^*$ $(i = 1, \ldots, n)$. *Perform* $2n$ *push phases. It follows from Definition* 2.8 *part* 2 *that* $x_i^{\min} = \underline{x}_i^n$ *and* $x_i^{\max} = \overline{x}_i^n$. *This procedure runs in* $O(mn)$ *time and is used in Megiddo's algorithm* [15].

For example, the TVPI system of Fig. 2 has $[\underline{x}^*, \overline{x}^*] = [-1, 1] = [x^{\min}, x^{\max}]$, and $[\underline{y}^*, \overline{y}^*] = [0, 2] = [y^{\min}, y^{\max}]$. The TVPI system of Fig. 3 has $[\underline{x}^*, \overline{x}^*] = [\underline{z}^*, \overline{z}^*] = [-\infty, +\infty]$, $[\underline{y}^*, \overline{y}^*] = [y^{\min}, y^{\max}] = [0, 1]$, and $[x^{\min}, x^{\max}] = [z^{\min}, z^{\max}] = [0, 2]$.

The following key observation is due to Shostak.

PROPOSITION 2.10 [17]. *If the system is feasible, then for all* $1 \leq i \leq n$, *the interval* $[x_i^{\min}, x_i^{\max}]$ *is the projection of the set of solutions on the* x_i *axis. Otherwise, for some* $1 \leq i \leq n$, $x_i^{\min} > x_i^{\max}$.

The proof follows from considering the possible structure of minimal sets of inequalities that imply a bound on a variable. Proposition 2.10 and the procedure described in Remark 2.9 assert that in order to find a feasible solution it suffices to consider all *simple* cycles in the associated graph. Shostak presented an exponential time algorithm that essentially examines all directed simple cycles [17].

The following corollary states that if two bounds are feasible separately but not simultaneously, then there exists a simple path that implies an inequality which asserts that the bounds are not simultaneously feasible.

COROLLARY 2.11. *Let* S *be a set of* TVPI *constraints and* $s_1 \in \{x_i \geq \alpha, x_i \leq \alpha\}$, $s_2 \in \{x_j \geq \beta, x_j \leq \beta\}$ *be two bounds. Let* $u_i = \overline{v}_i$ *if* s_1 *is an upper bound, and* $u_i = \underline{v}_i$ *otherwise*; *let* $u_j = \overline{v}_j$ *if* s_2 *is an upper bound, and* $u_j = \underline{v}_j$ *otherwise. Suppose* $S \cup \{s_1\}$, *and* $S \cup \{s_2\}$ *are feasible systems, but* $S \cup \{s_1, s_2\}$ *is not feasible. Under the above conditions the following holds. There exists a simple path* p *from* u_i *to* u_j^{-1} *such that* (i) $f_p(\alpha) < \beta$, *if* s_2 *is a lower bound, and* (ii) $f_p(\alpha) > \beta$, *if* s_2 *is an upper bound. Moreover, if* p *is the tightest* (*simple*) *path from* u_i *to* u_j^{-1} *relative to* $x_i = \alpha$, *then if* s_2 *is an upper* (*resp.*, *lower*) *bound,* $x_j = f_p(\alpha)$ *is the smallest* (*resp.*, *largest*) *value of* x_j *subject to* $S \cup \{s_1\}$.

Proof. Suppose that both s_1, s_2 are upper bounds (the proof for the other cases is similar). Consider $a_j \equiv x_j^{\min}$ under the system $S \cup \{s_1\}$ and $a_j' \equiv x_j^{\min}$ under the system S. We have $a_j' \leq \beta$ and $a_j > \beta$. By definition, $x_j^{\min}$ is either implied (a) by a cycle or a cycle and a path (i.e., implied only by inequalities that involve exactly two variables), or (b) a (possibly empty) directed simple path (i.e., by a single variable inequality). Since the system S contains the same two variable inequalities as $S \cup \{s_1\}$, a_j is determined by a system of type (b) consisting of the bound s_1 and a simple path p.

2.4. Characterizations of TVPI polyhedra. Proposition 2.10 relates the bounding box of the feasible region to the associated graph. We give several characterizations for polyhedra that comprise sets of solutions of TVPI systems (TVPI *polyhedra*). Polyhedra that cannot be expressed as such are *non*-TVPI *polyhedra*.

PROPOSITION 2.12. *Consider a polyhedron* $P \subset R^n$. *The following statements are equivalent*:

1. *The polyhedron* P *is a* TVPI *polyhedron.*

2. *For a bounded convex set $P' = P \cap B'$, where B' is a box, denote $a_i = \min_{x \in P'} x_i$, $b_i = \max_{x \in P'} x_i$ $(i = 1, \ldots, n)$, and denote by $B = \text{X}_{i=1}^{n}[a_i, b_i]$*[1] *the tightest bounding box for P'. Every P' as above is such that $\frac{1}{2}\boldsymbol{a} + \frac{1}{2}\boldsymbol{b} \in P'$.*
3. *Every set S of bounds (i.e., conditions of the form $x_i \geq \alpha$, $x_i \leq \alpha$, $x_i = \alpha$) has the following property. Denote by $B_S \in R^n$ the set of all feasible vectors for S. $P \cap B_S = \emptyset$ if and only if there exists a set $S' \subset S$, where $|S'| \leq 2$ such that $P \cap B_{S'} = \emptyset$.*
4. *Same as property 3, but S contains only equalities.*

In analysis done later in this paper we make use of the fact that property 1 implies 3. The other characterizations are presented for the sake of interest and completeness.

Proof. We first show that property 1 implies properties 2–4.

1 ⇒ 2 Was proved by Lueker, Megiddo, and Ramachandran [14].

1 ⇒ 3 Suppose P is a TVPI polyhedron. Let S_P be a set of TVPI constraints that define P and let $P_i = [\hat{x}_i^{\min}, \hat{x}_i^{\max}]$ $(i = 1, \ldots, n)$ be the corresponding projections. Let S be a set of bounds such that the combined TVPI system $S_P \cup S$ is infeasible. Note that the associated graphs of S_P and $S_P \cup S$ are identical. Denote by $x_i^{\min}, x_i^{\max}$ $(i = 1, \ldots, n)$ the bounds defined by the system $S \cup S_P$. By definition, each of $x_i^{\min}, x_i^{\max}$ $(i = 1, \ldots, n)$ that does not coincide with the respective end point of the interval P_i, is determined by a single bound from S and a simple path. Proposition 2.10 implies that the system $S_P \cup S$ is infeasible if and only if for some variable x_i, $x_i^{\min} > x_i^{\max}$. Hence at least one of the two bounds $x_i^{\min}, x_i^{\max}$ does not coincide with an end point of P_i. Consider the bounds $S' \subset S_P \cup S$ $(|S'| \leq 2)$ that determine $x_i^{\min} > x_i^{\max}$. Obviously, the system $S_P \cup S'$ is infeasible.

3 ⇒ 4 Obvious.

It remains to prove that either one of properties 2–4 implies property 1. Note that properties 1–4 are invariant under scalings and translations of coordinates.

Figure 4 provides an example of a non-TVPI polyhedron. It is easy to see that for this polyhedron property 2 does not hold (consider the bounding box $0 \leq x \leq 1$, $0 \leq y \leq 1$, $0 \leq z \leq 1$), and property 4 does not hold (consider $S = \{x = 0, y = 0, z = 0\}$).

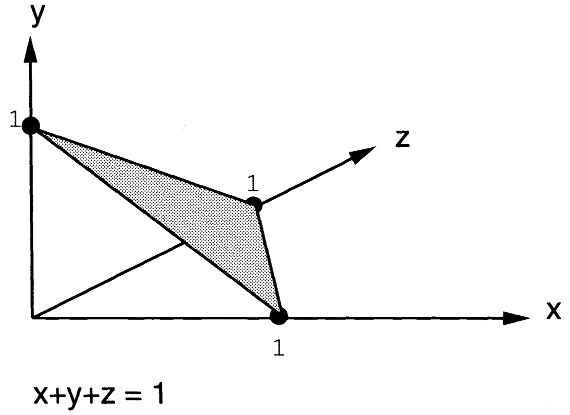


FIG. 4. *An example of a non-*TVPI *polyhedron.*

Assume that P is a non-TVPI polyhedron. If P is full dimensional, there exists at least one face F such that the equation of the hyperplane H constituting the affine hull of F involves

[1] $\text{X}_{i=1}^{n}[a_i, b_i]$ denotes the Cartesian product of the intervals $[a_i, b_i]$ $(i = 1, \ldots, n)$.

more than two variables. Without loss of generality (by permuting and scaling coordinates) we can assume that the equation is $\sum_{i=1}^{\ell} x_i = c$ (where $\ell \geq 3$ and $c \in R$), and for all points $\boldsymbol{x} \in P$, $\sum_{i=1}^{\ell} x_i \leq c$. Note that $P \cap H$ is full dimensional in H. Define $\hat{\boldsymbol{x}}$ to be some point in rel int $P \cap H$. If P is not full dimensional, consider a set of equations defining the affine hull of P. Consider a representation of P by the equalities $x_i = f_i(x_1, \ldots, x_{n'})$ $(i \geq n')$ for some $n' < n$ and a set of inequalities on $x_1, \ldots, x_{n'}$. Denote by $P' \subset R^{n'}$ the polyhedron defined by these inequalities. The representation is chosen such that P' is full dimensional. If one of the equations involves more than two variables assume (by permuting and scaling coordinates) that it has the form $x_\ell = -\sum_{i=1}^{\ell-1} x_i + c$. Define $(\hat{x}_1, \ldots, \hat{x_\ell})$ to be some point in rel int P'. If none of the equations involves more than two variables, then at least one face F of P' involves more than two variables. Using similar considerations to the ones used above, we assume without loss of generality that the equation defining the affine hull of F is $\sum_{i=1}^{\ell} x_i = c$, and for all points $\boldsymbol{x} \in P$, $\sum_{i=1}^{\ell} x_i \leq c$. Note that $P' \cap H$ is full dimensional in H. Define $\hat{\boldsymbol{x}}$ to be some point in rel int $P' \cap H$.

4 ⇒ 1 By definition of $\hat{\boldsymbol{x}}$, there exists a small enough $\epsilon \geq 0$ such that for $i = 1, \ldots, \ell$, there exists $\boldsymbol{x}^i \in P$ such that $x_i^i = \hat{x}_i - (\ell - 1)\epsilon$, and $x_j^i = \hat{x}_j + \epsilon$ $(j = 1, \ldots, i-1, i+1, \ldots, \ell)$. Consider the set of equations $S = \{x_j = \hat{x_j} + \epsilon \mid j = 1, \ldots, \ell\}$. Obviously, there is no point in P that satisfies S. On the other hand, there is a point that satisfies every subset of $\ell - 1 \geq 2$ constraints. Hence, property 4 does not hold for P.

2 ⇒ 1 By definition of $\hat{\boldsymbol{x}}$, there exists a small enough $\epsilon \geq 0$ such that for $i = 1, \ldots, \ell$, there exists $\boldsymbol{x}^i \in P$ such that $x_i^i = \hat{x}_i + (\ell - 1)\epsilon$, and $x_j^i = \hat{x}_j - \epsilon$ $(j = 1, \ldots, i-1, i+1, \ldots, \ell)$. Consider the box B defined by the intervals $[\hat{x}_j - \epsilon, \hat{x}_j + (\ell - 1)\epsilon]$ $(i = 1, \ldots, \ell)$. Since $\boldsymbol{x}^i \in B \cap P$, B is a bounding box for $B \cap P$. However, there is no $\boldsymbol{x} \in P$ for which $x_j = \hat{x}_j + (\ell - 2)\epsilon/2$ $(j = 1, \ldots, \ell)$. It follows that property 2 does not hold for P.

2.5. Classifying and locating values. We discuss two procedures that are used later in the paper and are based on repetitive applications of push phases and examining essential paths.

One of the procedures solves the following problem.

PROBLEM 2.13 [*Locate a value*]. *For a given* TVPI *system, a number* ξ, *and a variable* x_i, *locate* ξ *with respect to the interval* $[x_i^{\min}, x_i^{\max}]$, *that is, decide whether* (i) $\xi_i < x_i^{\min}$, (ii) $\xi_i > x_i^{\max}$, *or* (iii) $\xi_i \in [x_i^{\min}, x_i^{\max}]$.

We refer to solving Problem 2.13 as *locating* the value ξ_i. Aspvall and Shiloach [2], [3] proved the following.

PROPOSITION 2.14. *Problem* 2.13 *can be solved in* $O(mn)$ *operations.*

Aspvall and Shiloach utilized this result to solve TVPI systems: By solving instances of Problem 2.13, their algorithm conducts a binary search that finds a point $\xi_i \in [x_i^{\min}, x_i^{\max}]$ $(1 \leq i \leq n)$. They showed that finding such a point ξ_i can be done in $O(mnI)$ operations where I is the number of bits in the binary representation of the input. This yielded their $O(mn^2I)$ algorithm for TVPI systems: The algorithm performs n steps, where in step i a point $\xi_i \in [x_i^{\min}, x_i^{\max}]$ is found, and the equation $x_i = \xi_i$ is added to the system. If the original system is feasible, then $\boldsymbol{\xi} \in F$.

In §3 we present a $\tilde{O}(mn^2)$ deterministic algorithm for solving TVPI systems, that is based on locating values in the time bounds stated in Proposition 2.14. The material contained in the remainder of the current section is needed for achieving the randomized sequential bound of $\tilde{O}(n^3 + mn)$ and parallel time bounds of $\tilde{O}(n)$, and hence, it is not needed for the reading of §3. In the rest of the section we first characterize the possible structures of minimal subsystems that certify the infeasibility of values ("*certificates*"). We then classify infeasible values according to the types of their certificates. We also present a generic algorithm that performs a sequence of push phases for a given set of initial values. This algorithm maintains essential paths and tests for cycles in these paths. We apply this algorithm to solve both the value location problem and the *reveal* problem that is defined later.

Certificates of infeasibility. Suppose an interval $I = [a, b]$ is infeasible with respect to x_i (equivalently, either $[-\infty, b]$ or $[a, \infty]$ is infeasible). It follows from Proposition 2.10 that there exists a vertex w, a simple path p from $u \in \{\underline{v}_i, \overline{v}_i\}$ to w, and a disjoint simple cycle c starting at w such that the bound on w implied by c and the path p produce a bound on x_i that does not intersect I. Note that the path p and the cycle c comprise a minimal subsystem that implies the infeasibility of I. Also note that the cycle c may be empty, in which case the infeasibility follows from the path p and a bound on the variable associated with w. We refer to such a system as a *certificate* for I. If c is a closed cycle we say that the certificate is *closed*. If c is open or empty, we say that the certificate is *open*. When we refer to a certificate, we interchangeably mean the set of edges $E' = p \cup c$, the set E'^{-1} of the reversed edges, or the corresponding set of inequalities.

Classifying values. We classify values of variables according to the types of their certificates. Consider an interval I with respect to x_i. If I has a closed certificate we say that I is *strongly infeasible.* If only open certificates exist, we say that I is *weakly infeasible.* Otherwise, if no certificate exists, $I \cap [x_i^{\min}, x_i^{\max}] \neq \emptyset$ and we say that I is feasible. In particular these definitions apply when I consists of a single point. For $1 \leq i \leq n$, we denote by $[x_i^{\min *}, x_i^{\max *}]$ the set of all feasible and weakly infeasible values of x_i. Note that $x_i^{\min *}$, $x_i^{\max *}$ $(1 \leq i \leq n)$ are independent of the single variable inequalities.

We motivate distinguishing strongly and weakly infeasible values. We shall see that weakly infeasible values can be identified by applying push phases (at most $2n$ times) until for some $1 \leq i \leq n$ we have $\underline{x}_i > \overline{x}_i$. Strongly infeasible values are trickier to identify. Merely applying push phases is not enough. We need to keep track of essential paths, detect cycles in these paths, and compute the corresponding bounds.

We extend the definitions of the concepts above from intervals to values at vertices of G: A property (having a closed/open certificate, weak/strong infeasibility) holds for a value ξ at a vertex $\underline{v}_i$ (resp., $\overline{v}_i$) if it holds for the interval $[\xi, \infty]$ (resp., $[-\infty, \xi]$) with respect to x_i.

We present two corollaries of Propositions 2.5 and 2.10. Consider a feasible TVPI system and the associated graph. Suppose a sequence of push phases is performed.

COROLLARY 2.15. *If the values at the vertices are initially consistent (see Definition 2.8), they remain consistent after any number of push operations.*

COROLLARY 2.16. *Consider a vertex $u = \underline{v}_i$ (resp., $u = \overline{v}_i$) and a value $\xi = \underline{x}_i^{\ell}$ (resp., $\xi = \overline{x}_i^{\ell}$) for some ℓ. Let (ξ_j, v_j) $(j = 1, \ldots, k \leq \ell)$, where $(\xi_k, v_k) \equiv (\xi, u)$, be the pairs comprising the essential path (see Definition 2.6) associated with (ξ, u). If ξ is infeasible (resp., strongly infeasible) at u, then each of the pairs (ξ_j, v_j) $(1 \leq j < k)$ is such that ξ_j is an infeasible (resp., strongly infeasible) value of v_j.*

Consider an essential path that contains a closed cycle (a vertex appears more than once). The following proposition enables us to extract information from the cycle. It states that either all values following the start pair of the cycle are consistent (the path does not add information) or all values preceding the last pair of the cycle are strongly infeasible (an infeasible value is detected). Moreover, by considering the updating cycle we can determine which of the two situations occurs.

PROPOSITION 2.17. *Let ξ be a value at $u = \overline{v}$ (resp., $u = \underline{v}$). Suppose a closed cycle c, that starts and ends at u, is such that $f_c(\xi) < \xi$ (resp., $f_c(\xi) > \xi$). Then, either* (i) *ξ is strongly infeasible at u and the cycle c is a closed certificate, or* (ii) *if $F \neq \emptyset$, ξ is consistent at u.*

Proof. The bound implied by the cycle c is $f_c(y) \geq y$ (resp., $f_c(y) \leq y$). It suffices to show that it holds either (i) only for values y such that $y > \xi$ (resp., $y < \xi$), and hence ξ is strongly infeasible, or (ii) only for values y such that $y \leq \xi$ (resp., $y \geq \xi$) and hence, ξ is consistent. The function $f_c(y) - y$ is linear and thus can change sign only once. It cannot be a nonpositive (resp., nonnegative) constant, since this contradicts $f_c(\xi)$ being tighter than ξ. If the function is a negative (resp., positive) constant, the cycle implies that the system is infeasible (and therefore ξ is contradicted by the cycle). Otherwise, consider the value y^* such that $f_c(y^*) = y^*$. The bound implied by c is either $[-\infty, y^*]$ or $[y^*, \infty]$. Since ξ does not satisfy the bound inequality, it is not in this interval.

The following algorithm is applied to a set of values $\underline{x}_i^0, \overline{x}_i^0$ (for $1 \leq i \leq n$). The algorithm performs a sequence of push phases, and keeps track of essential paths. When the algorithm detects a nonsimple essential path, it either terminates or discards the path. If the algorithm terminates as a result of such a path, a closed certificate is found for one of the initial values. A vertex v is *active* at a particular point in the execution of the algorithm if the current value at v resulted from an essential path that is not discarded.

ALGORITHM 2.18 [*generic applications of push phases*].

1. *For* $i = 1, \ldots, n$, *initialize the values at* $\underline{v}_i$ *and* $\overline{v}_i$ *as* $\underline{x}_i^0$ *and* $\overline{x}_i^0$, *respectively.*
2. *For* $k = 1, \ldots, 2n$ *do steps* 3–7. *If the algorithm did not terminate, conclude that none of the initial values is strongly infeasible, and stop.*
3. *Perform a push phase. Denote the values at the vertices after the kth push phase by* $\underline{x}_j^k, \overline{x}_j^k$ $(1 \leq k \leq 2n)$.
   Keep track of all essential push operations that result from active vertices. If there are no such essential pushes or no active vertices, stop and conclude that none of the initial values is strongly infeasible.
   (*We assume that ambiguities about the edge that carried the essential push are resolved consistently according to some ordering on the edges.*)
4. *If* $k \neq 2^j$ *for all integers* j *and* $k \neq 2n$, *go to step* 3 (*next iteration*).
5. *Optional: If for some* j, $\underline{x}_j^k > \overline{x}_j^k$, *then stop.*
6. *For each active vertex* v, *let* p_v *be the essential path that corresponds to the last tightening of the value at* v.
   If p_v *contains a closed cycle* (*i.e., some vertex appears in more than one pair*), *execute the following subroutine* (step 7). *After considering all vertices, go to step* 3 (*next iteration*).
7. **Subroutine: Examine p_v.**
   Consider the last simple closed cycle c *on* p_v. *Denote by* u *the vertex where* c *starts and ends. Denote by* ξ *the value associated with* u *at its first occurrence on* c. *Check whether the bound implied by* c *contradicts* ξ. *If there is no contradiction,* ξ *is consistent* (*see Proposition* 2.17), *discard all essential paths originating from the first pair* (u, ξ) *of the cycle* c (*all correspond to consistent values according to Corollary* 2.15); *return. Otherwise, if* c *contradicts* ξ, ξ *is strongly infeasible. It follows from Corollary* 2.16 *that so are all values along the path* p_v *prior to the pair* $(u, f_c(\xi))$ (*the last occurrence of* u *on* p_v). *In particular, the initial value at the first vertex* w *of* p_v *is strongly infeasible. Terminate the execution of the algorithm.*

Complexity. The underlying computation of the algorithm is the Bellman-Ford single-source shortest-path computation [8], where we maintain information needed to construct the paths and test for cycles.

The algorithm terminates after $\ell \leq 2n$ iterations, in which case, it requires $O(m\ell)$ sequential time, and $O(\ell)$ expected time (or, $O(\ell \log\log m)$ deterministic time) using $O(m\ell)$ work on a CRCW PRAM.

If the algorithm terminates at step 7, it finds a closed certificate for the initial value at w. If the algorithm terminates during the ℓth iteration, the closed certificate found is of size $\theta(\ell)$.

PROPOSITION 2.19. *Consider an execution of Algorithm* 2.18, *where step* 5 *is skipped.*

1. *At least one of the initial values is strongly infeasible if and only if the algorithm terminates at step* 7.
2. *Suppose that exactly one of the initial values, ξ at the vertex w, is strongly infeasible. The algorithm terminates at step* 7 *with the same closed certificate regardless of the initial values at other vertices.*

Proof. We first prove part 1. The "if" direction is immediate. Suppose an initial value ξ at a vertex w has a closed certificate consisting of a path p of length $|p|$ to u_0 and a cycle c of length r that starts and ends at u_0. After at most $|p|$ iterations, the value at u_0 is at least as tight as $f_p(\xi)$ and hence, has a closed certificate consisting of c. Similarly, after $|p| + r$ iterations the same holds for all the vertices of c. Let $e_0, \dots, e_r$ and $u_0, \dots, u_r$, respectively, be the edges and vertices of c. We claim that all iterations for which $k \geq |p|$ result in an update of at least one of the values at $u_0, \dots, u_r$. Assume the claim is true. Assume to the contrary that the algorithm did not terminate at step 7. Therefore, it performs $2n$ iterations. It follows that at least one of the paths p_{u_j} $(1 \leq j \leq r)$ is of length $2n$, and hence, contains a closed cycle. Since the value at u_j is strongly infeasible, it follows from Corollary 2.16 that w or another vertex with a strongly infeasible initial value is the first vertex of p_{u_j}. It follows from Proposition 2.17 that each cycle in p_{u_j} contradicts the value at the start vertex. Hence, the algorithm terminates at step 7.

What remains is to prove the claim. Assume the contrary. Let $\xi_0, \dots, \xi_r$ be the respective values at $u_0, \dots, u_r$. In particular, none of the edges e_k $(0 \leq k \leq r)$ had an essential push. Hence, for all $j = 1, \dots, r$, $f_{e_j}(\xi_j)$ is not tighter than $\xi_{j+1 \bmod r}$. It follows that $f_{e_1 \cdots e_r}(\xi_0) = f_c(\xi_0)$ is no more tight than ξ_0. This is a contradiction for c being a closed certificate for ξ_0.

We prove part 2. Consider an initialization where ξ at w is the only strongly infeasible value. It follows from part 1 that the algorithm terminates in step 7, and finds an essential path p that starts at w and terminates in a closed cycle. All values along the path are strongly infeasible. Suppose that an initial path starting at a vertex other than w updates a value at a vertex in p. It follows from Corollary 2.16 that the updated value cannot be tighter or as tight as the value resulting from p. The proof follows. □

DEFINITION 2.20. *Suppose ξ is a strongly infeasible value at a vertex w. Consider an execution of Algorithm* 2.18 *where step* 5 *is skipped and ξ at w is the only strongly infeasible initial value* (*see Proposition* 2.19 *part* 2). *We refer to the unique closed certificate found by such an execution as* the certificate *of ξ.*

REMARK 2.21. *Consider a run of the algorithm where step* 5 *is performed.*

1. *Suppose that the algorithm terminates at step* 5, *where it detects $\underline{x}_j^k > \overline{x}_j^k$. Consider the initial pairs (u, ξ) and (w, μ) of the two essential paths that determined $(\underline{v}_j, \underline{x}_j^k)$ and $(\overline{v}_j, \overline{x}_j^k)$. It is not necessarily true that one of the initial pairs is infeasible. We can conclude, however, that the two corresponding bounds cannot be satisfied simultaneously by a feasible vector.*
2. *Consider a run of the algorithm where at least one of the initial values has an open certificate of size ℓ that contains a nontrivial cycle. The following is immediate: The algorithm terminates at either step* 7 *or step* 5 *within ℓ iterations.*

Locating a value. We apply Algorithm 2.18 to solve Problem 2.13.

Consider a value ξ_i. We show how to decide whether or not $\xi_i > x_i^{\max}$ (the case $\xi_i < x_i^{\min}$ is similar).

ALGORITHM 2.22 [*Locate a value with respect to* $x_i^{\max}$]. *Perform a run of Algorithm* 2.18, *where step* 5 *is enabled, for the following input of values*: $\underline{x}_j^0 = a_j$ *for* $j \neq i$, $\overline{x}_j^0 = b_j$ *for* $j = 1, \ldots, n$, *and* $\underline{x}_i^0 = \xi_i$.

Conclude as follows:

- *If the algorithm stopped at step* 3 (*no active vertices*), *determine that* $\xi_i \leq x_i^{\max}$.
- *If the algorithms stopped at step* 5, *consider the essential paths associated with* $\underline{x}_j^k > \overline{x}_j^k$. *If in neither path, the initial pair is* $(\underline{v}_i, \xi_i)$, *conclude that the system is infeasible. Otherwise, conclude that* $\xi_i > x_i^{\max}$.
- *Suppose the algorithm terminated at step* 7. *If* $w = \underline{v}_i$, *determine that* $\xi_i > x_i^{\max}$. *Otherwise, if* $w \neq \underline{v}_i$, *the system is infeasible.*
- *If the algorithm terminated at step* 2, *determine that* $\xi_i \leq x_i^{\max}$.

The correctness follows immediately from Proposition 2.19, part 1, and Remark 2.21.

Reveal a strongly infeasible value. We discuss applying Algorithm 2.18 to solve the following problem.

PROBLEM 2.23 [*reveal a strongly infeasible value*]. *Given are values* ξ_i, $a_i \leq \xi_i \leq b_i$ $(i \in I)$ *for the respective variables* x_i $(i \in I)$, *where* $I \subset \{1, \ldots, n\}$. *Do one of the following*:

1. *Conclude that all the values are feasible or weakly infeasible.*
2. *Find a strongly infeasible value* ξ_j *and a closed certificate.*

ALGORITHM 2.24 [*reveal*]. *Perform a run of Algorithm* 2.18, *where step* 5 *is skipped, for the input values:* $\overline{x}_i = \underline{x}_i = \xi_i$ $(i \in I)$, *and* $\overline{x}_i = b_i$, $\underline{x}_i = a_i$ $(i \notin I)$.

The following is a corollary of Proposition 2.19.

COROLLARY 2.25. *Algorithm* 2.24 *solves Problem* 2.23. *If none of the initial values is strongly infeasible, the algorithm requires* $O(mn)$ *operations. Otherwise, the algorithm terminates after* $\ell \leq 2n$ *iterations with a closed certificate of size* $\theta(\ell)$. *If exactly one of the values* ξ_i $(i \in I)$ *is strongly infeasible, the algorithm terminates with the same closed certificate regardless of the other values.*

3. The basic algorithm.

In this section we present a framework for solving TVPI systems. This framework allows us to state an algorithm for TVPI systems in terms of solving instances of Problem 2.13 (locating single values). The framework is stated in §3.1, and the correctness proof is given in §3.2. In §3.3 we reduce solving a TVPI system to locating $O(n(\log^2 n + \log m))$ values. Since locating a single value requires $O(mn)$ time ($O(n \log\log m)$ time using $O(mn)$ work on a CRCW PRAM), we obtain an $O(mn^2(\log^2 n + \log m))$ deterministic algorithm for solving TVPI systems (that runs in $\tilde{O}(n^2)$ time using $O(mn^2(\log^2 n + \log m))$ work on a CRCW PRAM). In §3.4 we introduce the problem of *locating a pool of values*. A key for further improvements is the reduction of solving a TVPI system to locating $O(\log^2 n + \log m)$ pools of values. A pool of values can be located naively by sequentially locating n single values. In §4 we present faster parallel and sequential algorithms for locating a pool, that yield better algorithms for solving TVPI systems.

3.1. The framework.

We first describe an idea introduced by Megiddo [15], which is the key in obtaining strongly polynomial time bounds. Consider the associated graph of some TVPI system. Every directed path in the associated graph corresponds to a two-variable inequality. Consider two inequalities that correspond to two paths between the same pair of vertices (v_i, v_j), where $v_i \in \{\overline{v}_i, \underline{v}_i\}$. These inequalities are linear, hence there exists a number a such that for all $xm_i \leq a$ one of the inequalities implies the other, and vice versa for $x_i \geq a$. If we focus only on feasible points $\boldsymbol{x}$ for which $x_i \geq a$ (similarly $x_i \leq a$), one of the inequalities is redundant. The algorithm eliminates paths and simultaneously restricts the feasible region. When "comparing" two paths, the decision about which one to eliminate is done as follows.

First, the number a, as above, is computed. The redundant path is determined by locating a with respect to feasible values of x_i (see Problem 2.13). Megiddo applied the above idea in an algorithm that basically performed n single-source shortest path Bellman-Ford type computations, where comparisons amount to locating values. The framework presented here is based on performing an all-pairs shortest path Floyd-Warshall [8] type computation that allows us to apply further ideas.

The following definition formalizes the concept of comparing paths.

DEFINITION 3.1.

1. *Suppose that p_1 and p_2 are two directed paths from u to v, and from v to w, respectively. Denote by $p_1 p_2$ the path from u to w obtained by concatenating p_1 and p_2.*
2. *Suppose that p_1 and p_2 are two paths from u to w. Let $I \subset R$ be an interval. We say that the path p_1 is at least as* tight *as p_2 relative to I (denote it by $p_1 \prec_I p_2$) if either $w \in \underline{V}$ and $f_{p_1}(\xi) \geq f_{p_2}(\xi)$ for all $\xi \in I$, or $w \in \overline{V}$ and $f_{p_1}(\xi) \leq f_{p_2}(\xi)$ for all $\xi \in I$.*
3. *Suppose $p_1, p_2, \ldots, p_k$ are paths from u to w. If for some i, $p_i \prec_I p_j$ for all j, we write $p_i = \min_{\prec_I}\{p_1, p_2, \ldots, p_k\}$. Note that when I is a single point, $\min_{\prec_I}$ is well defined.*

The algorithm maintains a set of intervals S_i $(1 \leq i \leq n)$, and a path p_{uw} from u to w for every pair of vertices (u, w). The algorithm runs in $\lceil \log_2 2n \rceil$ phases. During each phase, the paths and the intervals are considered for possible updates. Denote by S_i^k $(1 \leq i \leq n)$ and p_{uw}^k $(\{u, w\} \subseteq V)$ the intervals and paths, respectively, at the beginning of the kth phase. The algorithm has the following properties: (i) The set $\mathsf{X}_{i=1}^n S_i^k$ contains a feasible point, and (ii) the path p_{uw}^k is the tightest path from u to w, of length at most 2^k, relative to the interval S_i^k (where $u \in \{\overline{v}_i, \underline{v}_i\}$).

The algorithm is based on solving instances of the following problem.

PROBLEM 3.2. *Given are a graph G and a set of intervals $S_1, \ldots, S_n$ as in Definition 2.1. For every ordered pair $(u, v) \in V \times V$ of vertices we are given $p_{uv}^1, \ldots, p_{uv}^{k_{uv}}$, a collection of directed paths from u to v in G. The goal is to find a set of n intervals $I_1, \ldots, I_n$ and select a path $p_{uv}^* \in \{p_{uv}^1, \ldots, p_{uv}^{k_{uv}}\}$ for every pair (u, v) of vertices, where*

1. *$F \neq \emptyset \Rightarrow (\mathsf{X}_{i=1}^n I_i) \cap F \neq \emptyset$, and*
2. *$u \in \{\underline{v}_i, \overline{v}_i\} \Rightarrow p_{uv}^* = \min_{\prec_{I_i}}\{p_{uv}^1, \ldots, p_{uv}^{k_{uv}}\}$.*

We later suggest algorithms for Problem 3.2. The following algorithm uses it as a subroutine. It first initializes the tightest paths matrix by selecting the tightest edge out of every set of multiple edges (step 2). The rest of the algorithm consists of $\lceil \log 2n \rceil$ update phases of the tightest-paths matrix (step 4). During each phase, the algorithm considers n^2 sets of n paths (one for each pair of vertices). It then selects the tightest path in each set. In the last step, the tightest path matrix is used to compute a feasible vector.

ALGORITHM 3.3 [*Solve* TVPI *systems*].

1. *[Initialization] Construct S and G as in Definition 2.1.*
2. *[Initialize tightest paths matrix] For each pair of vertices consider the set of paths of length 1, i.e., all the multiple edges. Solve Problem 3.2 relative to these paths. For $(u, v) \in V \times V$: $p_{uv}^0 \leftarrow p_{uv}^*$. For $i = 1, \ldots, n$: $S_i \leftarrow S_i \cap I_i$, $S_i^0 \leftarrow S_i$.*
3. *For $k = 1, \ldots, \lceil \log_2 2n \rceil$, execute step 4. To continue, go to step 5.*
4. *For each pair $(u, v) \in V \times V$, consider the set of paths*

$$\left\{p_{uv}^{k-1}\right\} \cup \left\{p_{uw}^{k-1} p_{wv}^{k-1} | w \in V \setminus \{u, v\}\right\} .$$

Solve Problem 3.2 relative to these sets of paths.
For $(u, v) \in V \times V$: $p_{uv}^k \leftarrow p_{uv}^*$. For $i = 1, \ldots, n$: $S_i \leftarrow S_i \cap I_i$, $S_i^k \leftarrow S_i$.

5. *Denote* $p_{\underline{i}\,\underline{j}} \equiv p^{\lceil \log 2n \rceil}_{\underline{v}_i,\underline{v}_j}$, $p_{\underline{i}\bar{j}} \equiv p^{\lceil \log 2n \rceil}_{\underline{v}_i,\overline{v}_j}$, $p_{\bar{i}\underline{j}} \equiv p^{\lceil \log 2n \rceil}_{\overline{v}_i,\underline{v}_j}$, $p_{\bar{i}\bar{j}} \equiv p^{\lceil \log 2n \rceil}_{\overline{v}_i,\overline{v}_j}$ *(for* $1 \le i \le n$, $1 \le j \le n$*).*
   For $i = 1, \ldots, n$*:*
   $S_i \leftarrow S_i \cap \{x \mid f_{p_{\underline{i}\,\underline{i}}}(x) \le x\}$, $S_i \leftarrow S_i \cap \{x \mid f_{p_{\underline{i}\bar{i}}}(x) \ge x\}$, $S_i \leftarrow S_i \cap \{x \mid f_{p_{\bar{i}\underline{i}}}(x) \le x\}$.
6. *For* $i = 1, \ldots, n$, *compute intervals* $S'_i = [a'_i, b'_i]$ *as follows (where* $[a_j, b_j] = S_j$, *for* $1 \le j \le n$*):*

$$a'_i \leftarrow \max\left\{a_i, \max_j f_{p_{\bar{j}\underline{i}}}(b_j), \max_j f_{p_{\underline{j}\,\underline{i}}}(a_j)\right\},$$

$$b'_i \leftarrow \min\left\{b_i, \min_j f_{p_{\bar{j}\bar{i}}}(b_j), \min_j f_{p_{\underline{j}\bar{i}}}(a_j)\right\}.$$

7. *Compute a feasible solution* $\hat{\mathbf{x}}$ *as follows, for* $i = 1, \ldots, n$:

$$\hat{x}_i \in \left[\max\left\{a'_i, \max_{j<i} f_{p_{\bar{j}\underline{i}}}(\hat{x}_j), \max_{j<i} f_{p_{\underline{j}\,\underline{i}}}(\hat{x}_j)\right\}, \min\left\{b'_i, \min_{j<i} f_{p_{\bar{j}\bar{i}}}(\hat{x}_j), \min_{j<i} f_{p_{\underline{j}\bar{i}}}(\hat{x}_j)\right\}\right].$$

3.2. Correctness. We prove the following:

1. For $1 \le i \le n$, $S^0_i \supset S^1_i \supset \cdots \supset S^{\lceil \log_2 2n \rceil}_i \supset S'_i$.
2. If $\mathsf{X}^n_{i=1} S^0_i$ contains feasible points, then so do $\mathsf{X}^n_{i=1} S^k_i$ (for all $k > 0$) and $\mathsf{X}^n_{i=1} S'_i$.
3. p^k_{uw} ($u \in \{\underline{v}_i, \overline{v}_i\}$, $w \in V$) is the tightest path (relative to S^k_i) from u to w, of length less than or equal to 2^k.
4. If the system is feasible, the process described in step 7 results in a feasible vector $\hat{\mathbf{x}}$.

Claims 1 and 2 follow directly from the statement of the algorithm.

We prove Claim 3. Consider paths p_1, p_2 between the same pair of vertices. Assume they originate at $\{\underline{v}_j, \overline{v}_j\}$. It follows from the definition of $\prec$ that if $p_1 \prec p_2$ relative to S^k_j, then $p_1 \prec p_2$ relative to S^{k+1}_j. Consider paths of length at most 2^{k+1} from u to v. It suffices to show that when $p_{uv} = p_{uw}p_{wv}$ and $p'_{uv} = p'_{uw}p'_{wv}$, if $p_{uw} \prec_I p'_{uw}$ and $p_{wv} \prec_I p'_{wv}$, then $p_{uv} \prec_I p'_{uv}$. The latter is straightforward.

We prove Claim 4. We first show that $S'_i \subset [x^{\min}_i, x^{\max}_i]$ ($i = 1, \ldots, n$). We claim that $S^{\lceil \log 2n \rceil}_j \subset [\underline{x}^*_j, \overline{x}^*_j]$. If the latter holds, it follows from Proposition 2.10 that $S'_i \subset [x^{\min}_i, x^{\max}_i]$. Assume the contrary, that is, for some $1 \le j \le n$, $S^{\lceil \log 2n \rceil}_j \not\subset [\underline{x}^*_j, \overline{x}^*_j]$. Consider a point $\eta \in S^{\lceil \log 2n \rceil}_j \setminus [\underline{x}^*_j, \overline{x}^*_j]$. Assume that $\eta > \overline{x}^*_j$ (the case where $\eta < \underline{x}^*_j$ is similar). It follows that there exists a simple cycle c, such that either of the following is true:

1. The cycle c, starts and ends at $\overline{v}_j$, and is such that $f_c(\eta) < \eta$.
2. The cycle c, starts at $\underline{v}_j$ and ends at $\overline{v}_j$, and is such that $f_c(\eta) < \eta$.

Denote $c' = p_{\bar{j}\bar{j}}$ (for case (i)) and $c' = p_{\underline{j}\bar{j}}$ (for case (ii)). It follows from claim 3 that the cycle c' is the tightest simple cycle relative to $S^{\lceil \log 2n \rceil}_j$. Therefore, $f_{c'}(\eta) \le f_c(\eta)$. On the other hand, due to step 5 of the algorithm $f_{c'}(\eta) \ge \eta$. This is a contradiction. Thus we have shown that $S'_i \subset [x^{\min}_i, x^{\max}_i]$ ($i = 1, \ldots, n$).

We conclude the proof of Claim 4. Consider the computation performed in step 7. For $i = 1, \ldots, n$ consider the set of intervals $\hat{S}^i_\ell$ ($1 \le \ell \le n$) defined as follows. If $\ell < i$, $\hat{S}^i_\ell = \{\hat{x}_\ell\}$. Otherwise,

$$\hat{S}^i_\ell = \left[\max\left\{a'_\ell, \max_{j<i} f_{p_{\bar{j}\underline{\ell}}}(\hat{x}_j), \max_{j<i} f_{p_{\underline{j}\,\underline{\ell}}}(\hat{x}_j)\right\}, \min\left\{b'_\ell, \min_{j<i} f_{p_{\bar{j}\bar{\ell}}}(\hat{x}_j), \min_{j<i} f_{p_{\underline{j}\bar{\ell}}}(\hat{x}_j)\right\}\right].$$

We claim that for $i = 1, \ldots, n$ the following holds. For each $\mu \in \hat{S}^i_j$ there exists a feasible vector ξ, such that $\xi_j = \mu$ and $\xi_\ell \in \hat{S}^i_\ell$ ($\ell \ne j$). To conclude the proof of property 4, it suffices to prove the claim. We prove the claim by induction on i.

In the base case $\hat{S}^1_\ell = S'_\ell$ $(1 \le \ell \le n)$. We show that for every $\xi_j \in S'_j$ there exists a feasible solution $\boldsymbol{x}$ such that $x_j = \xi_j$ and $x_\ell \in S'_\ell$ for $\ell \ne j$. Assume the contrary. It follows from Proposition 2.12 that there exist two bounds $s_1 \in \{x_j \ge \xi_j, x_j \le \xi_j\}$ and $s_2 \in \{x_\ell \le b'_\ell, x_\ell \ge a'_\ell\}$ such that the system subject to $x_k \in S'_k$ $(1 \le k \le n)$ with the additional bounds s_1, s_2 is infeasible. Assume that s_1 is $x_j \ge \xi_j$ and s_2 is $x_\ell \le b'_\ell$ (the other cases are treated similarly). It follows from Corollary 2.11 that there exists a simple path p from $\overline{v}_\ell$ to $\overline{v}_j$ such that $f_p(b'_\ell) < \xi_j$. In contrast, it follows from property 3 and the computation of step 6 that $f_{p'}(b'_\ell) \ge b'_j$ for all paths p' from $\overline{v}_\ell$ to $\overline{v}_j$, hence a contradiction.

We prove the correctness of the induction step. Consider the step that determines $\hat{x}_i$. Assume that the claim is true for previous steps. The induction hypothesis asserts that subject to the constraints $x_\ell \in \hat{S}^i_\ell$ $(1 \le \ell \le n)$, we have $[x^{\min}_\ell, x^{\max}_\ell] = \hat{S}^i_\ell \equiv [\hat{a}^i_\ell, \hat{b}^i_\ell]$. For $1 \le \ell \le n$, let $\tilde{x}^{\min}_\ell$, $\tilde{x}^{\max}_\ell$ be the respective values of $x^{\min}_\ell$, $x^{\max}_\ell$ subject to $x_\ell \in \hat{S}^i_\ell$ $(1 \le \ell \le n)$ and the additional constraint $x_i = \hat{x}_i$. We need to show that for $1 \le \ell \le n$, $[\tilde{x}^{\min}_\ell, \tilde{x}^{\max}_\ell] = \hat{S}^{i+1}_\ell$. The direction $\hat{S}^{i+1}_\ell \supset [\tilde{x}^{\min}_\ell, \tilde{x}^{\max}_\ell]$ is obvious. Consider $\tilde{x}^{\max}_j$ (the arguments for $\tilde{x}^{\min}_j$ are similar). If $\tilde{x}^{\max}_j = \hat{b}^i_j$ we are done. Otherwise, it follows from Proposition 2.10 that there exists a simple path p from $v_i \in \{\overline{v}_i, \underline{v}_i\}$ to $\underline{v}_j$ such that $f_p(\hat{x}_i) = \tilde{x}^{\max}_j$. It follows from property 3 that either $p' = p_{\bar{i}\underline{j}}$ or $p' = p_{\underline{i}\underline{j}}$ is as tight as p. Hence $f_{p'}(\hat{x}_i) = \tilde{x}^{\max}_j = \hat{b}^{i+1}_j$.

REMARK 3.4. *Note that step 7 can be replaced by choosing* $\hat{x}_i = (a'_i + b'_i)/2$, $1 \le i \le n$. *This follows from the fact that for the system subject to* $x_i \in S'_i$ $1 \le i \le n$, *we have* $[x^{\min}_i, x^{\max}_i] = S'_i$ $(1 \le i \le n)$, *and from Proposition* 2.12 *part* 2.

3.3. Complexity of the naive implementation. The complexity is dominated by the calls to an algorithm for Problem 3.2. We present a naive algorithm for Problem 3.2 that is based on locating single values (see Problem 2.13). The algorithm consists of n sequential stages as follows. At the ith stage the interval I_i is computed and a tightest path is found for each of the $O(n)$ sets of paths that emanate from either one of $\{\underline{v}_i, \overline{v}_i\}$. In the proceeding stages we consider the system with the additional constraint $x_i \in I_i$.

For $1 \le i \le n$, denote by $k^{(i)} = \sum_u (k_{\overline{v}_i u} + k_{\underline{v}_i u})$, the number of paths emanating from $\{\underline{v}_i, \overline{v}_i\}$. We discuss stage i. We consider $O(n)$ sets of paths with the goal of choosing a tightest path in each set. This is done in $O(\log k^{(i)})$ iterations, where in each iteration the total number of paths that need to be considered reduces from r to $n + 3(r - n)/4$. Initially, $r = k^{(i)}$. After the stage terminates, $r = n$ and each set contains a single path (the tightest path). Each iteration is as follows. First, we pair up paths that belong to the same pair of vertices. Each such pair corresponds to a comparison between the two paths that needs to be resolved. For each pair, we compute the intersection of the two linear functions that correspond to the two paths. Each "comparison" amounts to locating the intersection point with respect to the interval $[x^{\min}_i, x^{\max}_i]$. We solve one instance of Problem 2.13 to locate the median of these $r/2$ intersections. By doing this, half the comparisons are resolved, and the number of remaining paths is at most $n + 3(r - n)/4$.

The following proposition is immediate. The parallel bounds follow from known parallel selection algorithms (see e.g., [13]).

PROPOSITION 3.5. *Stage i can be performed by an* $O(k^{(i)} + mn \log k^{(i)})$ *algorithm. A parallel algorithm runs in* $O(n \log m \log\log m \log k^{(i)})$ *determinisitc time* (*alternatively, by allowing randomization, in* $O(n \log k^{(i)})$ *expected time*) *using* $O(mn \log k^{(i)} + k^{(i)})$ *work on a CRCW PRAM.*

We discuss the resulting complexity of Algorithm 3.3. In step 2, the total number of paths $(\sum_{i=1}^n k^{(i)})$ is the number of edges in the graph. Hence the number of operations is $O(mn^2 \log m)$. In step 4, $k^{(i)} = O(n^2)$ $(1 \le i \le n)$. Hence, the number of operations in each execution of step 4 is $O(n^3 + mn^2 \log n)$. Step 4 is performed $\lceil \log_2 2n \rceil$ times. It follows that the total number of operations is $O\left(mn^2(\log m + \log^2 n)\right)$. On a CRCW PRAM the algorithm runs in $O\left(n^2(\log m + \log^2 n)\log m \log\log m\right)$ time (alternatively, by allowing randomization, in $O\left(n^2(\log m + \log^2 n)\right)$ expected time), using $O(mn^2(\log m + \log^2 n)$ work.

3.4. Solving TVPI systems by locating pools of values.

Consider the "naive" algorithm for solving Problem 3.2. The algorithm consists of n stages that are performed sequentially. We present a different algorithm that performs the stages "concurrently," where iterations of different stages are interleaving. The algorithm consists of $O(\max_{1 \le i \le n} \log k^{(i)})$ phases, where at phase i all the ith iterations of the n stages are performed.

An iteration of stage $1 \le j \le n$ amounts to locating a value of x_j. Performing a phase amounts to solving an instance of the following problem.

PROBLEM 3.6 [*Locate a pool of values*]. *Given are a graph G and a set of intervals $S_1, \ldots, S_n$ as in Definition* 2.1. *Also given are a set of values $\xi_i \in S_i$ $(i \in I)$ for the corresponding variables x_i $(i \in I)$, where $I \subset \{1, \ldots, n\}$. The goal is to find a set of intervals J_i $(i \in I)$, such that* (i) $F \neq \emptyset \Rightarrow F \cap \{x \in R^n | \bigwedge_{i \in I} x_i \in J_i\} \neq \emptyset$, *and* (ii) $\xi_i \notin$ interior J_i $(i \in I)$.

We give a more elaborate description of the algorithm that reduces Problem 3.2 to locating pools of values. The correctness is straightforward. The algorithms performs interleaving executions of the "n" stages. This is done in $O(\log \max_i k^{(i)})$ phases. Denote by $I \subset \{1, \ldots, n\}$ the set of stages that did not terminate at the current phase (initially, $I = \{1, \ldots, n\}$). Phase j is as follows:

1. For each $\ell \in I$: compute the value ξ_ℓ of x_ℓ arising from an iteration of stage ℓ. This computation takes $O(n + (3/4)^j \sum_{i \in I} k^{(i)})$ time, and takes, with optimal speedup, $O(\log \max_i k^{(i)} \log\log \max_i k^{(i)})$ deterministic time and constant expected time on a CRCW PRAM.
2. Locate the pool (solve Problem 3.6) ξ_ℓ $(\ell \in I)$. Let J_ℓ $(\ell \in I)$ be the intervals constituting the solution.
3. For $\ell \in I$, $S_\ell \leftarrow S_\ell \cap J_\ell$.

It follows that Problem 3.2 is reduced to solving $O(\max_i \log k^{(i)})$ instances of Problem 3.6 and $O(n \max_i \log k^{(i)} + \sum_{1 \le i \le n} k^{(i)})$ additional time. The problem of solving TVPI systems is therefore reduced to solving $O(\log^2 n + \log m)$ instances of Problem 3.6 and $O(m + n^3 \log n)$ additional computation. Note that, in parallel, the additional computation can be performed in $O(\log m \log\log m + \log^2 n \log\log n)$ time deterministically and in $O(\log n)$ expected time with optimal speedup.

Problem 3.6 can be solved naively by sequentially solving $|I|$ instances of Problem 2.13 (locating a single value). This requires mn^2 time, and $\tilde{O}(n^2)$ time in parallel. In §4 we give algorithms that improve over this bound.

4. Algorithms for locating a pool.

The problem of solving TVPI systems was reduced to locating $O(\log^2 n + \log m)$ pools (Problem 3.6) and $O(n^3 \log n + m)$ additional computation.

In §3.4 we showed that a pool can be located in mn^2 time, and $\tilde{O}(n^2)$ time in parallel. In this section we discuss two algorithms for locating a pool. In §4.1 we present an algorithm that runs in $O(mn^2)$ sequential time, and $\tilde{O}(n)$ time in parallel with optimal speedup. In §4.2 we overview a randomized $O(mn \log^3 n)$ expected time algorithm. A parallel implementation runs in $\tilde{O}(n)$ expected time using $O(m)$ processors. The details of the randomized algorithm are given in §§5 and 6.

4.1. $\tilde{O}(n)$ time using $O(n^2 m)$ work.

In this subsection we prove the following.

PROPOSITION 4.1. *Problem* 3.6 *can be solved on a CRCW PRAM in* $O(n \log\log m)$ *time* (*alternatively, by allowing randomization, in* $O(n)$ *expected time*) *using* $O(mn|I|)$ *work.*

COROLLARY 4.2. *Algorithm* 3.3 *has a parallel implementation on a CRCW PRAM that runs in* $O\left(n \log\log m(\log^2 n + \log m)\right)$ *time* ($O\left(n(\log^2 n + \log m)\right)$ *expected time if randomization is allowed*) *and performs* $O(mn(\log^2 n + \log m))$ *work.*

The following algorithm solves Problem 3.6 within the time bounds stated in Proposition 4.1.

ALGORITHM 4.3 [*Locate a pool*].

```
1. In parallel, locate (solve Problem 2.13) each of the values $\xi_i$ $(i \in I)$.
   For $i \in I$ do as follows:
     • If $\xi_i \geq x_i^{\max}$, determine $J_i \leftarrow \{z | z \leq \xi_i\}$.
     • If $\xi_i \leq x_i^{\min}$, determine $J_i \leftarrow \{z | z \geq \xi_i\}$.
   Let $I' \subset I$ be the set of indices such that $x_i^{\min} < \xi_i < x_i^{\max}$ $(i \in I')$.
2. For each $i \in I'$ perform the following computation:
   Initialize the values at the vertices of $G$ to $\overline{x}_i = \underline{x}_i = \xi_i$ and $\overline{x}_j = \infty$, $\underline{x}_j = -\infty$
   $(j \neq i)$. Apply $2n$ push phases. For $1 \leq j \leq n$ denote the final values at the nodes
   $\underline{v}_j$, $\overline{v}_j$, respectively, by $\underline{x}_j^i$, $\overline{x}_j^i$.
3. Construct a graph $H$ as follows: The graph $H$ has $|I'|$ nodes $w_i$ $(i \in I')$. There is
   an edge between $w_i$ and $w_j$ if and only if $\xi_j < \underline{x}_j^i$ or $\xi_j > \overline{x}_j^i$.
4. Compute a maximal independent set (corresponds to nodes $I^+ \subset I'$) in $H$.
   Choose intervals $J_i$ $(i \in I')$ as follows:
   If $i \in I^+$, $J_i \leftarrow \{\xi_i\}$.
   If $i \in I' \setminus I^+$, then
     • If $\min_{j \in I^+} \overline{x}_i^j < \xi_i$, $J_i \leftarrow \{z | z \leq \xi_i\}$.
     • If $\max_{j \in I^+} \underline{x}_i^j > \xi_i$, $J_i \leftarrow \{z | z \geq \xi_i\}$.
```

We prove the correctness of the algorithm. It follows from Corollary 2.11 that $F \cap \{x | x_i = \xi_i \wedge x_j = \xi_j\} \neq \emptyset$ if and only if $\underline{x}_j^i \leq \xi_j \leq \overline{x}_j^i$. Hence the graph H captures the dependencies between pairs of values. Proposition 2.12 (see equivalence of properties 1 and 4) implies that a set of single variable equations is feasible if and only if every pair is feasible. Hence, the set of intervals J_i $(i \in I')$ solves Problem 3.6.

4.2. Overview of a $\tilde{O}(mn)$ algorithm. In this subsection we present an overview of a faster randomized algorithm for locating a pool of values. A single value can be located in $O(mn)$ operations by using Algorithm 2.22. In the previous section we solved Problem 3.2 more efficiently by inferring from locating a value of one variable about other values of the same variable. To locate a set I of values of different variables, however, we still needed $O(mn|I|)$ operations. The randomized approach presented in this section enables us to infer about values of other variables as well, and in particular, locate a pool of values within $\tilde{O}(mn)$ operations.

We prove the following.

THEOREM 4.4. *Problem* 3.6 *can be solved sequentially in an expected number of* $O(mn \log^3 n)$ *operations and on a CRCW PRAM in* $O(n \log^3 n)$ *expected time using* $O(m)$ *processors.*

Consequently, TVPI systems can be solved in an expected number of

$$O\left(n^3 \log n + mn(\log^5 n + \log m \log^3 n)\right)$$

operations.

Consider a pool of values $\xi_1, \ldots, \xi_n$ for the corresponding variables. These values are classified into three groups as follows (see §2.5):

1. $\xi_i \in [x_i^{\min}, x_i^{\max}]$ (ξ_i is feasible).

2. $\xi_i \notin [x_i^{\min}, x_i^{\max}]$, but $\xi_i \in [x_i^{\min *}, x_i^{\max *}]$ (ξ_i is weakly infeasible).
3. $\xi_i \notin [x_i^{\min *}, x_i^{\max *}]$ (ξ_i is strongly infeasible).

We propose a two-stage algorithm for Problem 3.6. In the first stage the algorithm locates all the strongly infeasible values ξ_i $(i \in I')$, and determines whether $\xi_i < x_i^{\min *}$ or $\xi_i > x_i^{\max *}$. The respective intervals are determined to be $J_i = [-\infty, \xi_i]$ if $\xi_i > x_i^{\max *}$ and $J_i = [\xi_i, \infty]$ if $\xi_i < x_i^{\min *}$. In the second stage the algorithm solves an easier special case of Problem 3.6 where the values are guaranteed to be either feasible or weakly infeasible. In §5 we present an algorithm that solves the first stage and in §6 we present an algorithm for the second stage. These algorithms run in $O(mn \log^3 n)$ time, and in $O(n \log^3 n)$ parallel time using $O(m)$ processors on a CRCW PRAM.

By combining the above results we obtain algorithms for Problem 3.6 with the running times stated in Theorem 4.4.

5. Locating the strongly infeasible values.

We present an algorithm for the following problem.

PROBLEM 5.1 [*Determine the strongly infeasible values*]. *Given are values ξ_i $(i \in I)$ for the corresponding variables x_i $(i \in I)$, where $I \subset \{1, \dots, n\}$. The goal is to determine for each $i \in I$ whether $\xi_i < x_i^{\min *}$, $\xi_i > x_i^{\max *}$, or $\xi_i \in [x_i^{\min *}, x_i^{\max *}]$.* We prove the following.

PROPOSITION 5.2. *Problem* 5.1 *can be solved* (i) *sequentially, in an expected number of $O(mn \log^3 n)$ operations, and* (ii) *on a CRCW PRAM, in $O(n \log^3 n)$ expected time, using $O(m)$ processors.*

For purposes of analysis we classify the strongly infeasible values of variables according to the sizes of their certificates (see Definition 2.20).

DEFINITION 5.3. *A strongly infeasible value ξ_i of a variable x_i is ℓ* big *if the certificate is of size at most ℓ. A value is $(\ell_1, \ell_2]$ big if it is ℓ_2 big but not ℓ_1 big. We interchangeably refer to strongly infeasible values as* big. *Note that all strongly infeasible values are $2n$ big.*

We explain the motivation for this classification. The algorithm that determines all the big values is based on a tradeoff between the following two properties. These properties are stated more formally and proved later. The first one "favors" big values with small certificates: For any ℓ-big value ξ it takes $O(m\ell)$ operations (by using "reveal") to find the certificate. Hence, a decision procedure that locates an ℓ-big value requires only $O(m\ell)$ operations. The second property favors values with large certificates: We introduce the procedure "check" that considers a big value ξ along with an associated certificate E'. The "check" procedure can on average locate many other big values whose certificates intersect E'.

5.1. The algorithm.

The description of the algorithm includes calls to the following three procedures:

- The first procedure "reveal" (Problem 2.23) considers a set of k values, and either finds a big value that belongs to the set or concludes that all these values are feasible or weakly infeasible. Algorithm 2.24 solves "reveal." Recall (see Corollary 2.25) that the algorithm takes $O(mn)$ time. If the input set contains big values, the algorithm terminates after $O(m\ell)$ operations (where $\ell \leq n$) with a closed certificate of size $\theta(\ell)$. If the original set contained exactly one big value ξ_j, the certificate found by the algorithm is the certificate of ξ_j.
- The second procedure "check" considers an ℓ-big value and the corresponding certificate. The "check" procedure uses the certificate to produce upper and lower bounds on the feasible regions of other variables. These bounds enable us to make decisions regarding other values in the pool. An $O(m\ell)$ time "check" algorithm is given in §5.3. In §5.4 we discuss properties of the "check" algorithm. We show that at least half of the $(\ell, 2\ell]$-big values in the pool have the following property: When a "check"

is applied to any of them, it determines at least $\ell/(6n) - 1$ of the other $(\ell, 2\ell]$ big values in the pool.

- The third procedure considers the set of unlocated values I and computes a crude estimate r^* to the number of big values r. The estimate r^* is such that $1/2 \leq r^*/r \leq 2$ with probability at least $1/2$. In §5.5 we present an "estimate" algorithm that performs $O((\log\log n)^2)$ calls to the "reveal" procedure. The time complexity is dominated by these calls and hence is $O(mn(\log\log n)^2)$.

The following algorithm determines all big values in a pool of $|I| = \hat{n}$ values. The algorithm performs iterations where each iteration determines some of the big values. The set I and the number $\hat{n}$ are updated accordingly. Denote by $r \leq \hat{n}$ the (unknown) number of big values in I. Appropriate values for the constants C_1, C_2, and C are given later.

ALGORITHM 5.4 [*Determine all the big values*].

◇ *Loop A: Steps* 1–9

1. *Apply the "reveal" procedure to the set I. If there are no big values, stop.*
2. *Compute an estimate* r^* *for* r *such that with probability at least* $1/2$, $1/2 \leq r^*/r \leq 2$. *Reset the following two counters:* $tt \leftarrow 0$ [*number of operations*]; $tb \leftarrow 0$ [*number of big values discarded*] (*in the current iteration of Loop A*).

◇ *Loop B: Steps* 3–9

3. *If either* (i) $tb \geq r^*/4$ (*successful iteration of Loop A*), *or* (ii) $tt \geq Cmn\log^2 n$ (*unsuccessful iteration of Loop A*), *go to step* 1 (*next iteration*).
4. $b \leftarrow 0$ [*the number of big values discarded in the current iteration of Loop B*].
5. *If* $r^* \geq \hat{n}/4$, $s \leftarrow 1$. *Otherwise,* $s \leftarrow \lfloor \hat{n}/r^* \rfloor$. *Choose* $k = \lceil C_1 \log n \rceil$ *random samples* $S_1, \ldots, S_k$ *of size* s (*with replacements*) *from the pool* I.
6. *Execute "concurrently"* k *runs of "reveal"* (*see Algorithm* 2.24) *applied to the sets* $S_1, \ldots, S_k$ *as follows*:
7. *Initialize copy* i *according to the set of values* S_i. *Set* $\ell \leftarrow 0$ [*current phase number*]. *Set* $K \leftarrow \{1, \ldots, k\}$ [*set of "active" runs*].

◇ *Loop C: Steps* 8–9

8. *Perform an additional iteration* (*push phase*) *to the runs in* K. *Set* $\ell \leftarrow \ell + 1$. *Let* $K' \subset K$ *be the* (*possibly empty*) *subset of the runs for which certificates are found. Apply "check" operations to these certificates, and discard from* I *all the big values that are determined. Increment* b *and* tb *accordingly, and set* $K \leftarrow K \setminus K'$.
9. *If either* $\ell = 2n + 1$ *or* $b \geq \lceil C_2 \ell r^*/(n \log n) \rceil$ *then* $tt \leftarrow tt + C_1 \ell m \log n$, *go to step* 3 [*next iteration of Loop B*]. *Otherwise, go to step* 8.

In §5.2 we prove the following.

PROPOSITION 5.5. *There exist constants* C_1,C_2, *and* C *as follows. If the estimate* r^* *is such that* $1/2 \leq r^*/r \leq 2$, *then with probability at least* $1/2$, *the current iteration of Loop A terminates after determining* $r^*/4$ *big values* (*the iteration is successful*).

It follows that with probability at least $1/4$, each iteration of Loop A determines at least $1/8$ of the big values in the pool. Hence, the expected number of iterations performed until all big values are determined is $O(\log n)$. Each iteration of Loop A runs in $O(mn\log^2 n)$ time. Hence, the expected time in which Algorithm 5.4 terminates is $O(mn\log^3 n)$. This concludes the proof of Proposition 5.2.

REMARK 5.6. *Note that the computation of the estimate* r^* *in step* 2 *can be replaced by running Loop A with* $r^* = 2^1, 2^2, \ldots, 2^{\lfloor \log \hat{n} \rfloor}$. *This results in an additional* $\log n$ *factor in the worst-case time bound.*

5.2. Probability for a successful iteration of Loop A.

In this subsection we prove Proposition 5.5.

Denote by r_i ($i = 1, \dots, \lceil \log 2n \rceil$) the number of $(2^{i-1}, 2^i]$ big values in I.

DEFINITION 5.7. *A sample $S \subset I$ of values from the pool is* good *if the following three conditions hold*:

1. *The sample contains exactly one big value ξ_j. Denote the size of the certificate of ξ_j by $2^{i-1} < \ell \leq 2^i$.*
2. *ξ_j determines* (*using "check"*) *at least $2^{i-1} r_i/(6n)$ big values in I* (*including ξ_j itself*).
3. $r_i \geq r/(4\lceil \log 2n \rceil)$.

We claim that property (iii) holds for at least 3/4 of the big values. Let

$$L = \{1 \leq i \leq \lceil \log 2n \rceil \mid r_i < r/(4\lceil \log 2n \rceil)\} .$$

We need to show that $\sum_{i \in L} r_i < r/4$. The claim follows from the inequality

$$\sum_{i \in L} r_i/r < \sum_{i \in L} 1/(4\lceil \log 2n \rceil) = |L|/(4\lceil \log 2n \rceil) < 1/4 .$$

It follows from Theorem 5.17 in §5.4 that property (ii) holds for at least half the $(2^{i-1}, 2^i]$ big values (for all i). Hence, properties (ii) and (iii) hold for at least 3/8 of the big values.

The following proposition motivates the definition of good samples.

PROPOSITION 5.8. *Consider an application of the reveal algorithm to a good sample, followed by a check to the certificate found. There exists a number $\ell = 2^i$ ($1 \leq i \leq \lceil \log 2n \rceil$), such that by using $O(m\ell)$ operations we can locate at least $\lceil \ell r/(24n\lceil \log 2n \rceil)\rceil$ big values.*

Proof. Denote by ξ_j the big value in the sample, and let $\ell = 2^i$ be such that the size of the certificate of ξ_j is in $(2^{i-1}, 2^i]$. Consider an application of the reveal algorithm to the sample. It follows from Proposition 2.19, part 2 that the certificate is found within $O(m\ell)$ operations. A check procedure applied to the certificate locates at least $2^{i-1} r_i/(6n) - 1$ additional big values from the pool. Note that $r_i \geq r/(4\lceil \log 2n \rceil)$. Hence at least $\ell r/(24n\lceil \log 2n \rceil)$ big values are located.

PROPOSITION 5.9. *Consider an integer $s \geq 1$ such that* (i) $s = 1$ *if $r \geq \hat{n}/2$ and in some cases, when $r \geq \hat{n}/16$, and* (ii) *otherwise, $c = rs/\hat{n}$ is such that $0.25 \leq c \leq 2$. Consider a random sample $S \subset I$* (*with replacements*) *of s elements. In some cases, and only when $s - 1 \geq 4$, the random sample contains $s - 1$ elements. There exists a constant ρ_1 such that with probability at least ρ_1, S contains exactly one big value.*

Proof. If $|S| = s$, the probability that S contains exactly one big value is

$$P = s\frac{r}{\hat{n}}\left(1 - \frac{r}{\hat{n}}\right)^{s-1} .$$

If $s = 1$, then $16r \geq \hat{n}$ and hence $P \geq 1/16$. Otherwise, $2r < \hat{n}$, hence $c/s < 1/2$. We obtain that

$$P \geq c\left((1 - c/s)^{s/c}\right)^c (1 - c/s)^{-1} \geq c/4^c \geq 1/32 .$$

If $|S| = s - 1$, we have

$$P = (s - 1)\frac{r}{\hat{n}}\left(1 - \frac{r}{\hat{n}}\right)^{s-2} \geq ((s - 1)/s)c/4^c \geq (4/5)(1/32) .$$

Consider an iteration of Loop A. During iterations of Loop B, the number r of big values in the set I decreases, but the value of the estimate r^* is not updated. The iteration of Loop A terminates when $r^*/4$ big values are discovered. Therefore, if initially $1/2 \leq r^*/r \leq 2$, then during the iterations of Loop B we have $1/2 \leq r^*/r \leq 4$.

COROLLARY 5.10. *Suppose that* $1/2 \le r^*/r \le 4$, *and consider a random sample* $S \subset I$ (*with replacements*) *of size* s, *where* $s = 1$ *when* $r^* \ge \hat{n}/4$ *and* $s = \lfloor \hat{n}/r^* \rfloor$ *otherwise. The sample* S *is good with probability* $\rho \ge 3\rho_1/8$.

COROLLARY 5.11. *Let* $C_1 = -1/\ln(1-\rho)$. *Consider* $\lceil C_1 \log n \rceil$ *randomly chosen samples* (*with replacements*) *of size* s. *If* $1/2 \le r^*/r \le 4$, *the probability that none of the samples is good is smaller than* $1/n$.

Proof. The probability that no sample is good is $p = (1-\rho)^{\lceil C_1 \log n \rceil}$. It follows that $p \le 1/n$ when $C_1 \ge -1/\ln(1-\rho)$.

Assume that initially $1/2 \le r^*/r \le 2$. Choose $C_2 = 1/96$. We compute the expected number of steps performed until $r^*/4$ big values are located.

DEFINITION 5.12. *An iteration of Loop B is* beneficial *if* $b \ge \lceil C_2 \ell r^*/(n \log n) \rceil$ *when the iteration terminates* (*at* step 9).

PROPOSITION 5.13. *Consider an iteration of Loop B. If at least one of the samples is good, the iteration is beneficial.*

Proof. The proof follows from Proposition 5.8 and the fact that for $r^* \le 4r$,

$$\lceil C_2 \ell r^*/(n \log n) \rceil \le \lceil \ell r/(24n \lceil \log 2n \rceil) \rceil .$$

The counter tt is such that at the end of each iteration of Loop B, the running time so far of the current iteration of Loop A is $O(tt)$. A beneficial iteration increases tt in $C_1 m \ell \log n$ units (runs in time $O(C_1 m \ell \log n)$), and each unbeneficial iteration by $C_1 mn \log n$ units ($O(C_1 mn \log n)$ time). It follows from Proposition 5.13 and Corollary 5.11 that an expected number of $(n-1)/n$ of all iterations of Loop B are beneficial. Hence, the expected number of operations performed by unbeneficial iterations is not bigger than the expected number of operations performed by beneficial iterations.

To conclude the proof of Proposition 5.5, it suffices to count the expected number of operations performed by beneficial iterations (increase of the counter tt) until at least $r^*/4$ big values are located.

Denote by t_i ($i = 1, \ldots, \lceil \log n \rceil$) the number of beneficial iterations that ended within $(2^{i-1}, 2^i]$ phases. The total expected increase in the value of tt by beneficial iterations is

$$\Delta_{tt} \le C_1 \sum_{i=1}^{\lceil \log n \rceil} t_i 2^i m \log n .$$

The expected final value of tt (number of operations by beneficial and unbeneficial iterations) is $tt \le 2\Delta_{tt}$. At all iterations but the last we have

$$\sum_{i=1}^{\lceil \log n \rceil} (t_i C_2 2^{i-1} r^*/(n \log n)) \le r^*/4 .$$

Hence, after the last iteration, $\sum_{i=1}^{\lceil \log n \rceil} t_i 2^i \le n \log n/(2C_2) + n$. Therefore,

$$\Delta_{tt} \le ((C_1 + C_2)/(2C_2)) mn \log^2 n .$$

Thus, the expected final value of tt is

$$tt \le ((C_1 + C_2)/C_2) mn \log^2 n .$$

We choose $C = 2(C_1 + C_2)/C_2$. It follows that with probability $1/2$, at least $r^*/4$ of the big values are located before the value of tt reaches $Cmn \log^2 n$. This concludes the proof of Proposition 5.5.

5.3. The "check" procedure. In this subsection we give an algorithm for performing "checks." The "check" algorithm is applied to a $(2^{k-1}, 2^k]$-big value ξ_i and the certificate ξ_i, and provides information from which it can be concluded that certain other values are also big.

We explain how a closed certificate for a value of one variable is used to obtain information about other variables in the following remark.

REMARK 5.14. *Suppose ξ_i is a big value of x_i. The certificate G_i consists of a path from v_i to a vertex u and a cycle that starts and ends at u, such that $v_i = \underline{v}_i$ if $\xi_i > x_i^{\max *}$, and $v_i = \overline{v}_i$ if $\xi_i < x_i^{\min *}$. For a vertex w in G_i, define $\xi_i(w) = f_p(\xi_i)$, where p is the path in G_i from v_i to w. Define $\xi_i(v_i) = \xi_i$. It follows from Corollary 2.16 that for every vertex w of G_i, the value $\hat{\xi}_i(w)$ at w is strongly infeasible.*

The "check" algorithm is described below. The input is a $(2^{k-1}, 2^k]$-big value ξ_i of x_i along with the certificate G_i.

ALGORITHM 5.15 [*check*].

```
1. For each vertex $v \in G_i$, compute the big value $\xi(v)$ (as in Remark 5.14).
2. Initialize the values of $\overline{x}_1, \ldots, \overline{x}_n$ and $\underline{x}_1, \ldots, \underline{x}_n$ as follows. If $\overline{v}_j \in G_i$, set $\underline{x}_j = \xi_i(\overline{v}_j)$; otherwise, if $\overline{v}_j \notin G_i$, set $\underline{x}_j = -\infty$. If $\underline{v}_j \in G_i$, set $\overline{x}_j = \xi_i(\underline{v}_j)$; otherwise, if $\underline{v}_j \notin G_i$, set $\overline{x}_j = \infty$.
3. Perform $2^k$ push phases.
4. Make conclusions as follows. If for some value $\eta$ we have $\eta \geq \overline{x}_j$ (resp., $\eta \leq \underline{x}_j$), conclude that $\eta > x_j^{\max *}$ (resp., $\eta < x_j^{\min *}$).
```

The initial values at the vertices are consistent (see Remark 5.14), and hence (see Corollary 2.15), the final values are consistent. It follows that the conclusions made by the "check" algorithm are correct.

DEFINITION 5.16. *If in step 4 of Algorithm 5.15 we deduce that a value η is big, we say that ξ_i 2^k* locates *η.*

5.4. Properties of the "check" procedure. This subsection establishes the following theorem.

THEOREM 5.17. *Consider a collection of b $(2^{k-1}, 2^k]$-big values. At least half the values have the following property. If a "check" is applied to any of them, it results in 2^k locating at least $b2^{k-1}/(6n) - 1$ other values from the collection.*

The following proposition analyzes the dependencies among values of different variables.

PROPOSITION 5.18. *Let ξ_i and ξ_j be big values of x_i and x_j, respectively. Let the respective certificates G_i, G_j, and the nodes v_i, v_j be as in Remark 5.14. Suppose that $|G_i| \leq \ell$, $|G_j| \leq \ell$ for some ℓ. If the sets of vertices participating in G_i and G_j intersect, at least one of the values ξ_i and ξ_j is ℓ located by the other.*

Proof. Suppose $u \in \underline{V}$ (similar for $u \in \overline{V}$) participates both in G_i and in G_j. Let p_i be the directed path in G_i from v_i to u and let p_j be a directed path in G_j from v_j to u. (See Fig. 5 for an illustration, where $i = 1$ and $j = 2$.) We claim that if $f_{p_i}(\xi_i) \leq f_{p_j}(\xi_j)$, then ξ_i ℓ locates ξ_j; if $f_{p_j}(\xi_j) \leq f_{p_i}(\xi_i)$, ξ_j ℓ locates ξ_i. Assume that $f_{p_i}(\xi_i) \leq f_{p_j}(\xi_j)$, and consider an application of "check" to ξ_i. The "check" algorithm determines that $f_{p_i}(\xi_i)$ is a strongly infeasible value of u. The algorithm initializes u^{-1} to the consistent value $f_{p_i}(\xi_i)$ and applies ℓ push phases. Consider the path p_j^{-1} from u^{-1} to v_j^{-1}. Since $|p_j^{-1}| \leq \ell$, the value at v_j^{-1} after ℓ push phases must be at least as tight as $\xi_j^* \equiv f_{p_j^{-1}}(f_{p_i}(\xi_i))$. To conclude the proof, note that ξ_j^* is at least as tight as ξ_j at v_j^{-1}, since $\xi_j = f_{p_j^{-1}}(f_{p_j}(\xi_j))$ and we assumed that $f_{p_j}(\xi_j)$ is no more tight than $f_{p_i}(\xi_i)$ at u^{-1}. Hence ξ_i ℓ-locates ξ_j.

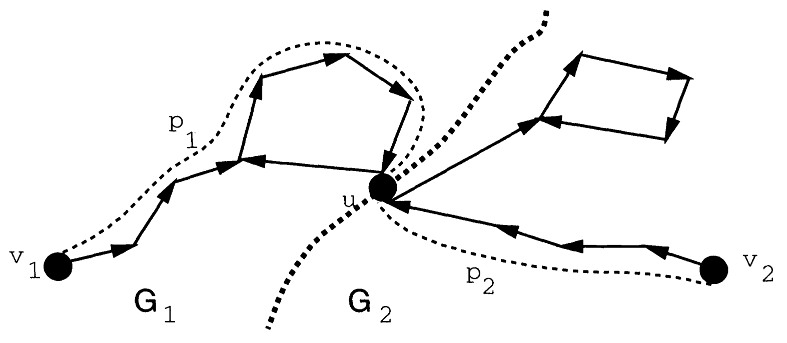


FIG. 5. *Dependence of intersecting certificates.*

DEFINITION 5.19. *Consider a pool of b* $(2^{k-1}, 2^k]$*-big values. The* influence *graph of this pool is defined as follows. Each big value in the pool corresponds to a vertex in the influence graph. If one value* 2^k *locates another, then there is a directed edge from the vertex of the former to the vertex of the latter.*

Theorem 5.17 is an immediate corollary of the following theorem.

THEOREM 5.20. *In the influence graph of a pool of b* $(2^{k-1}, 2^k]$*-big values, at least half of the vertices have an out degree greater than* $\frac{1}{6}b2^{k-1}/n - 1$. We first prove two propositions that are used in the proof of the theorem.

PROPOSITION 5.21. *In the intersection graph of b subsets of* $\{1, \ldots, n\}$ *with cardinality* ℓ*, the sum of the degrees is at least* $b^2\ell/n - b$.

Proof. Let s_i $(i = 1, \ldots, n)$ be the number of subsets containing i. Obviously, $\sum_{i=1}^n s_i = b\ell$. The sum of the degrees plus the number of vertices (subsets) is at least $\sum_{i=1}^n s_i^2/\ell$. This expression is minimized when $s_i \in \{\lfloor b\ell/n \rfloor, \lceil b\ell/n \rceil\}$ $(i = 1, \ldots, n)$. It follows that the sum of degrees is at most $\lceil b^2\ell/n \rceil - b$.

PROPOSITION 5.22. *In any orientation of the edges of the intersection graph of b subsets of* $\{1, \ldots, n\}$ *with cardinality* ℓ*, at least half the vertices have out degrees greater than* $\frac{1}{6}b\ell/n - 1$.

Proof. It follows from Proposition 5.21 that the sum of out degrees (i.e., the total number of edges) in any subgraph induced by b' vertices is at least $\frac{1}{2}(b'^2\ell/n - b')$. Suppose, to the contrary, that there exists a subset U of cardinality $|U| = b' = b/2$ where the out degree of each vertex is less than or equal to $\frac{1}{6}b\ell/n - 1$. The total number of edges (that equals the sum of out degrees) in the subgraph induced by these vertices is $\frac{1}{3}b'^2\ell/n - b'$, hence a contradiction since it is less that $\frac{1}{2}(b'^2\ell/n - b')$.

Proof of Theorem 5.20. The cardinality of a certificate of a $(2^{k-1}, 2^k]$-big value is greater than or equal to $\ell = 2^{k-1}$. It follows from Proposition 5.18 that if two big values have intersecting certificates, then in the influence graph there is an edge between the corresponding vertices. Hence, the influence graph contains an intersection graph of b subsets of $\{1, \ldots, n\}$ of cardinality ℓ. Thus, the proof follows from Proposition 5.22. □

5.5. Estimating the number of big values. In this subsection we prove the following proposition.

PROPOSITION 5.23. *An estimate* r^* *such that* $1/2 \le r^*/r \le 2$ *with probability at least* $1/2$*, can be found using* $O((\log\log n)^2)$ *"reveal" operations.* The problem of computing an estimate can be stated in the following terms. We are given a collection of $\hat{n}$ stones, where r of them are radioactive. The number r is not known. We are equipped with a primitive gauge that, when exposed to a set of stones, can determine if at least one of them is radioactive (i.e., "reveal" operation). A *good estimate* to the number of radioactive stones is a number r^* such that $1/2 \le r^*/r \le 2$. The goal is to compute a good estimate that is correct with probability at least 1/2, by using at most $O((\log\log\hat{n})^2)$ gauge readings. We first give some useful propositions.

PROPOSITION 5.24. *Denote by $p_0(s)$ the probability that a sample (with replacements) of size s contains no radioactive stones and let $c \equiv rs/\hat{n}$. Under these conditions,* (i) $c < \ln(1/p_0(s))$, *and* (ii) *if $r/\hat{n} \le 0.5$, $c \ge \log(1/p_0(s))/\log 4$.*

Proof.

$$p_0(s) = \left(1 - \frac{r}{\hat{n}}\right)^s = \left(\left(1 - \frac{r}{\hat{n}}\right)^{\frac{\hat{n}}{r}}\right)^c .$$

It follows that $p_0(s) < e^{-c}$ and if $r/\hat{n} \le 0.5$, $p_0 \ge 4^{-c}$.

COROLLARY 5.25. *Suppose $p_0(s)$ is known. We can conclude as follows:*

1. *If* $0.136 \le p_0(s) \le 0.5$, *then* $1/2 \le c \le 2$.
2. *If* $0.146 \le p_0(s) \le 0.49$, *then* $0.514 \le c \le 1.92$.
3. *If* $p_0(s) \le 0.156$, *then* $c \ge 1.34$.
4. *If* $p_0(s) \ge 0.48$, *then* $c \le 0.74$.

PROPOSITION 5.26. *Let B' be the random variable (r.v.) that corresponds to the number of non-radioactive samples out of $B = 5000(\lceil\log\log\hat{n}\rceil + 1)$ random samples of size s.*

$$\text{Prob}\left\{|B'/B - p_0(s)| \ge 0.01\right\} \le 1/(2\lceil\log\log\hat{n}\rceil + 2) .$$

Proof. B' is the number of successes of B Bernoulli trials with success probability $p_0(s)$, and hence, has a binomial distribution. Therefore B' has expected value $p_0(s)B$ and variance $Bp_0(s)(1 - p_0(s))$ (see [11] for background). It follows from Chebyshev inequality that $\text{Prob}\left\{|B' - p_0(s)B| \ge 0.01B\right\} \le 2500B/B^2 = 2500/B = 1/(2\lceil\log\log\hat{n}\rceil + 2)$.

Denote by $\boldsymbol{p}_0'(s)$ the r.v. B'/B, and by $p_0'(s)$ a value of $\boldsymbol{p}_0'(s)$.

Below we state an algorithm that computes the estimate r^*. For simplicity, we assume that $\hat{n}$ is a power of 2 (the general case can be handled by a small modification of the algorithm). The algorithm is based on performing a binary search on the $\log\hat{n} - 1$ possible estimate values $2^1, 2^2, \ldots, 2^{\log\hat{n}-1}$.

ALGORITHM 5.27 [*compute an estimate*].

```
1. If $p_0'(2) \le 0.35$, stop and return the estimate $r^* = 2^{\log\hat{n}-1}$.
2. Initialize $a = 1$, $b = \lfloor\log\hat{n}\rfloor$. Repeat step 3 until $a = b$. Return $r^* = \hat{n}/2^a$.
3. Set $j = \lceil(a + b)/2\rceil$, and $s = 2^j$.
     • If $0.146 \le p_0'(s) \le 0.49$, stop and return the estimate $r^* = \hat{n}/s$.
     • If $p_0'(s) < 0.146$, set $b = j$ and go to step 3.
     • If $p_0'(s) > 0.49$, set $a = j$ and go to step 3.
```

We show that with probability at least 0.5, $1/2 \le r^*/r \le 2$. Consider the first step. If $r \ge \hat{n}/2$, $p_0(2) \le 1/4$. Hence, $\text{Prob}\{p_0'(2) \le 0.35\} \ge 1 - 1/(2\lceil\log\log\hat{n}\rceil + 2)$. If $r < \hat{n}/4$, $p_0(2) > 9/16$. Hence, $\text{Prob}\{p_0'(2) < 0.35\} \le 1/(2\lceil\log\log\hat{n}\rceil + 2)$. It follows that if $r \ge \hat{n}/2$, the algorithm stops with a good estimate with probability at least $1 - 1/(2\lceil\log\log\hat{n}\rceil + 2)$. If $r \le \hat{n}/4$ the probability that the algorithm stops at Step 1 with a bad estimate is smaller than $1/(2\lceil\log\log\hat{n}\rceil + 2)$.

The algorithm performs at most $\lceil\log\log\hat{n}\rceil$ iterations of step 3. Each iteration terminates with a further restriction of the set $a, a+1, \ldots, b$. Consider a single iteration of step 3. Suppose that at the beginning of the iteration, at least one of the values $2^a, 2^{a+1}, \ldots, 2^b$ constitutes a good estimate. It follows from Proposition 5.26 that with probability $(1 - 1/(2\lceil\log\log\hat{n}\rceil + 2))$, $|p_0'(s) - p_0(s)| \le 0.01$. Assume that this is the case. Note that for $r^* = \hat{n}/s$ we have $c = r/r^*$. Therefore, it follows from Corollary 5.25 that when the iteration ends the modified interval $a, a + 1, \ldots, b$ is such that at least one of $2^a, 2^{a+1}, \ldots, 2^b$ is a good estimate.

Therefore, the probability that when the algorithm terminates 2^a is a good estimate is at least

$$(1 - 1/(2\lceil \log\log \hat{n}\rceil + 2))^{\lceil \log\log \hat{n}\rceil + 1} \geq 0.5 \,.$$

The algorithm performs $O(\log\log \hat{n})$ iterations, where each iteration computes an appropriate value of p_0'. Each iteration requires $O(\log\log \hat{n})$ "gauge readings" (see Proposition 5.26). Hence, the total number of "gauge readings" performed by the algorithm is $O((\log\log \hat{n})^2)$. This concludes the proof of Proposition 5.23.

6. Locating a pool of weakly infeasible values and feasible values. In this section we present an algorithm that asserts the following.

THEOREM 6.1. *Problem* 3.6, *where the values ξ_i $(i \in I)$ are guaranteed to be either feasible or weakly infeasible, can be solved* (i) *sequentially, in $O(mn \log^3 n)$ expected time, and* (ii) *on a CRCW PRAM, in $O(n \log^3 n)$ expected time, using $O(m)$ processors.*

The following is a key property of feasible and weakly infeasible values.

PROPOSITION 6.2. *Let ξ_i and ξ_j be feasible or weakly infeasible values of x_i and x_j, respectively. If there exists a path p from $v_j \in \{\underline{v}_j, \overline{v}_j\}$ to $v_i \in \{\underline{v}_i, \overline{v}_i\}$ such that $f_p(\xi_j)$ is at least as tight as ξ_i at the vertex v_i, there exists such a simple path p'.*

Proof. We assume that $v_j = \overline{v}_j$, $v_i = \overline{v}_i$. (The proof for the other cases is similar.) Consider the path p from v_j to v_i. Suppose a vertex u occurs more than once on p. Consider two of the occurrences of u. Let $p = p_1 p_2 p_3$, where p_1 is the prefix of p until the first occurrence and p_3 is the suffix of p starting from the second occurrence of u. If $f_{p_1 p_2}(\xi_j)$ is not tighter than $f_{p_1}(\xi_j)$ at u, the path $p' = p_1 p_3$ is at least as tight as p. Consider iterating the above process. It follows that there exists a path $\hat{p}$ from v_j to v_i that is at least as tight as p with respect to $x_j = \xi_j$, and when a vertex appears more than once, the value at subsequent occurrences is always tighter. We claim that each vertex may occur at most once in $\hat{p}$. Assume the contrary, and consider the first cycle c along $\hat{p}$. Let u be the vertex where c starts and ends, and let $p' = p_1 c$ be the corresponding prefix of $\hat{p}$. It follows from Proposition 2.17 that either $f_{p_1}(\xi_j)$ is strongly infeasible at u, or $f_{p_1}(\xi_j)$ is consistent at u. If $f_{p_1}(\xi_j)$ is strongly infeasible, then p' is a closed certificate for $[-\infty, \xi_j]$ with respect to x_j and we get a contradiction to the assumption that ξ_j is not strongly infeasible. Otherwise, $f_{p_1}(\xi_j)$ is consistent at u. It follows that for every prefix p_2 of $\hat{p}$ that contains p_1 and ends at a vertex u', the value $f_{p_2}(\xi_j)$ is consistent at u'. Consider the last cycle c' along $\hat{p}$, and let $p' = c' p_3$ be the corresponding suffix of $\hat{p}$. It follows that p' is a closed certificate for $[f_{\hat{p}}(\xi_j), \infty]$ at v_i^{-1}. Hence, since $\xi_i \geq f_{\hat{p}}(\xi_j)$, p' is a closed certificate for $[\xi_i, \infty]$. This is a contradiction to ξ_i not being strongly infeasible.

We present two procedures that are subroutines of the algorithm for locating a pool. The first procedure is applied to a subset of the values and returns a partial solution. The second procedure extends a partial solution as much as possible without restricting the feasible region further. We discuss the first procedure in detail. The input is a subset of values $\xi_i \in S_i$ $(i \in I')$, where $I' \subset I$. The procedure determines intervals J_i $(i \in I'')$, where $I'' \subset I'$ is a subset of the values in the input set. The intervals J_i $(i \in I'')$ constitute a partial solution for Problem 3.6, that is, there exists a set of intervals J_i $(i \in I \setminus I'')$ such that J_i $(i \in I)$ is a solution of Problem 3.6.

ALGORITHM 6.3 [*find a partial solution*].

```
1. Initialize the values at the vertices of $G$ as follows:
   $\overline{x}_i = \underline{x}_i = \xi_i$ $(i \in I')$, $\overline{x}_i = b_i$, $\underline{x}_i = a_i$ $(i \notin I')$,
2. Apply $2n$ push phases.
```

3. *For $i \in I'$ do as follows*:
   - *If* $\xi_i = \overline{x}_i = \underline{x}_i$, *set* $J_i = \{\xi_i\}$.
   - *If* $\xi_i = \overline{x}_i < \underline{x}_i$, *set* $J_i = [\xi_i, \infty]$.
   - *If* $\xi_i = \underline{x}_i > \overline{x}_i$, *set* $J_i = [-\infty, \xi_i]$.
   - *Otherwise,* J_i *is not determined.*

   Let $I'' \subset I'$ *be the set of all* $i \in I'$ *for which* J_i *is determined. For* $i \in I''$ *do:* $S_i \leftarrow S_i \cap J_i$.

Denote by F^* the feasible region prior to applying Algorithm 6.3. Consider an application of the algorithm. We show that if $F^* \neq \emptyset$ then the system remains feasible, that is, $F \neq \emptyset$. The following proposition proves a necessary condition for correctness: If $F^* \neq \emptyset$, then for all $i \in I''$, $F^* \cap \{x | x_i \in J_i\} \neq \emptyset$.

PROPOSITION 6.4. *Suppose* $\underline{x}_i = \xi_i$ *(resp.,* $\overline{x}_i = \xi_i$*) is weakly infeasible, and the certificate does not consist only from a bound and a simple path. There exists a simple path p from* $\underline{v}_i$ *to* $\overline{v}_i$ *(resp.,* $\overline{v}_i$ *to* $\underline{v}_i$*), such that* $f_p(\xi_i) < \xi_i$ *(resp.,* $f_p(\xi_i) > \xi_i$*).*

Proof. It suffices to prove the existence of a not necessarily simple path p with the above properties. The existence of a simple path would follow from Proposition 6.2. We assume $\underline{x}_i = \xi_i$ is weakly infeasible (the other case is proved similarly). There exists $1 \leq j \leq n$ and two simple paths emanating from $\underline{v}_i$, p_1 to $\overline{v}_j$ and p_2 to $\underline{v}_j$ such that $f_{p_1}(\xi_i) < f_{p_2}(\xi_i)$. Consider the path $p = p_1 p_2^{-1}$ from $\underline{v}_i$ to $\overline{v}_i$. We claim that $f_p(\xi_i) < \xi_i$. To prove the claim note that $f_p(\xi_i) = f_{p_2^{-1}} \circ f_{p_1}(\xi_i)$. Since the end vertices of p_2^{-1} both lie in $\underline{V}$, the function $f_{p_2^{-1}}$ is monotone increasing. Hence, $f_{p_2^{-1}} \circ f_{p_1}(\xi_i) < f_{p_2^{-1}} \circ f_{p_2}(\xi_i) = \xi_i$.

We show that the intervals J_i $(i \in I'')$ constitute a partial solution.

PROPOSITION 6.5. $F = F^* \cap \{x | \bigwedge_{i \in I''} x_i \in J_i\} \neq \emptyset$.

Proof. We first claim that J_i is feasible for all $i \in I''$. Consider $i \in I''$. If ξ_i is feasible the claim is immediate. Otherwise, assume to the contrary that $J_i = [\xi_i, \infty]$ (similar for $[-\infty, \xi_i]$) is weakly infeasible. If the certificate consists of a bound in S_j $(j \in I)$ and a simple path then, when the algorithm terminates, we have $\overline{x}_i < \xi_i$ (since the initialization is with values at least as tight as S_j $(j \in I)$). This is a contradiction. Otherwise, the certificate contains an open cycle. The contradiction follows from Proposition 6.4. This concludes the proof of the claim. In order to prove the proposition, Proposition 2.12, part 3 asserts that it suffices to show that for every pair $\{i, j\} \in I''$, $F^* \cap \{x | x_i \in J_i,\ x_j \in J_j\} \neq \emptyset$. The latter follows from Corollary 2.11.

We discuss the procedure that extends a partial solution. Suppose we applied Algorithm 6.3 and as a result, for $i \in I''$, we updated the intervals S_i and determined J_i. The following algorithm may determine the interval J_i for some additional values $i \in I \setminus I''$.

ALGORITHM 6.6 [*extend a partial solution*].

1. *Initialize the values at the vertices of G as follows.* $\underline{x}_i = a_i$ *and* $\overline{x}_i = b_i$ $(i \in I)$.
2. *Apply* $2n$ *push phases.*
3. *For* $i \in I \setminus I''$ *conclude as follows*:
   - *If* $\overline{x}_i \leq \xi_i$, *set* $J_i = \{-\infty, \xi_i\}$.
   - *If* $\underline{x}_i \geq \xi_i$, *set* $J_i = \{\xi_i, \infty\}$.

   Let $\hat{I}$ *be the set of all* $i \in I \setminus I''$ *such that* J_i *was determined.*
   For $i \in \hat{I}$ *do:* $S_i \leftarrow S_i \cap J_i$.

REMARK 6.7. *Proposition* 2.5 *asserts that the feasible region is not changed by the updates of the intervals* S_i *done by Algorithm* 6.6. *In particular, if the feasible region was nonempty prior to applying the algorithm it remains nonempty.*

6.1. The algorithm. The following algorithm considers feasible and weakly infeasible values ξ_i $(i \in I)$, and solves Problem 3.6 for these values. The solution consists of a set of intervals J_i $(i \in I)$. The algorithm performs iterations, where in each iteration some intervals J_i $(i \in I' \subset I)$ are determined. The intervals J_i $(i \in I')$ are incorporated into the system, we set $I \leftarrow I \setminus I'$, and the problem is reduced to locating the values ξ_i $(i \in I)$.

ALGORITHM 6.8 [*locate a pool*].

```
1. Repeat the following until $I = \emptyset$.
2. For $\ell = 1, 2, 4, \ldots, 2^{\lfloor \log |I| \rfloor}$, execute steps 3–5:
3. Choose (uniformly at random) $\ell$ elements from $I$. Let $I' \subset I$ be the set of chosen
   elements.
4. Apply Algorithm 6.3 to $I'$. The algorithm determines $J_i$ for $i \in I'' \subset I'$.
5. Apply Algorithm 6.6 and determine $J_i$ for additional values $\hat{I} \in I$. Set $I \leftarrow I \setminus$
   $\{I'' \cup \hat{I}\}$.
```

It follows from Proposition 6.5 and Remark 6.7 that if the problem was feasible, it remains feasible at any point during the execution of the algorithm. Hence, when the algorithm terminates, the intervals J_i $(i \in I)$ constitute a solution for Problem 3.6.

6.2. Complexity. For values ξ_i $(i \in I)$, consider a solution for Problem 3.6, that is, intervals J_i $(i \in I)$ such that $J_i \in \{[-\infty, \xi_i], [\xi_i, \infty]\}$ and $F \cap \{\boldsymbol{x} | \bigwedge_{i \in I} x_i \in J_i\} \neq \emptyset$. We designate vertices v_i $(i \in I)$ according to the choice of intervals J_i $(i \in I)$: If $J_i = [-\infty, \xi_i]$, $v_i = \overline{v}_i$. If $J_i = [\xi_i, \infty]$, $v_i = \underline{v}_i$.

We define a relation on the set I, with respect to the solution J_i $(i \in I)$:

DEFINITION 6.9. *Let $i \in I$, $j \in I$ be two elements.*

1. *We say that j* locates *i if there exists a path p from v_j to v_i such that $f_p(\xi_j)$ is at least as tight as ξ_i on v_i.*
2. *We say that i* interferes *with j if there exists a path p from v_i^{-1} to v_j^{-1} such that $f_p(\xi_i)$ is tighter than ξ_j on v_j^{-1}.*
3. *We define the relation $\prec$ on I as follows. $j \prec i$ if and only if i interferes with j. $j \preceq i$ if and only if j locates i. (Note that $j \prec i$ implies $j \preceq i$, and $j \preceq i \wedge i \npreceq j$ implies $j \prec i$.)*
4. *For an element $i \in I$, denote by* $\operatorname{pred}(i) = \{j \in I | j \preceq i\}$ *the set of elements that locate i, and by* $\operatorname{succ}(i) = \{j \in I | j \succ i\}$ *the set of elements that interfere with i.*

The following is a corollary of Proposition 6.2.

COROLLARY 6.10. *If a path p as in Definition* 6.9, *parts* 1 *and* 2 *exists, then there must exist a simple path with the same properties.*

The following proposition relates the locate and interfere relations to the algorithms presented earlier. The proof follows from Corollary 6.10.

PROPOSITION 6.11. *Let $i \in I$, $j \in I$ be two elements.*

1. *Suppose that j locates i. If $S_j \subset J_j$, then an application of Algorithm* 6.6 *would determine J_i.*
2. *Suppose that i interferes with j. Consider an application of Algorithm* 6.3 *where $\{i, j\} \subset I'$. The interval J_j will not be determined as a result of such a run. Moreover, if $j \in I'$, the interval J_j is determined in this run if and only if for all values $i \in I'$, i does not interfere with j.*

We characterize the values located in an iteration of Algorithm 6.8, in terms of pred and succ.

COROLLARY 6.12. *The following relations hold in an execution of Steps* 4–5: $I'' = \{k \in I' | \operatorname{succ}(k) \cap I' = \emptyset\}$ *and* $\hat{I} = \{k \in I \setminus I'' | \operatorname{pred}(k) \cap I'' \neq \emptyset\}$.

PROPOSITION 6.13. *The relation $\prec$ is a partial order on I.*

Proof. The transitivity of $\prec$ is immediate. We need to show that $\prec$ has no cycles. If there is a cycle, it follows from the transitivity that for some $j \in I$, j interferes with j. The path p is in this case a simple cycle that constitutes a closed certificate for ξ_j. This contradicts the assumption that ξ_j is not strongly infeasible.

Note that each iteration of steps 3–5 results in determining at least one interval, and therefore, the algorithm terminates within $O(n)$ iterations. In order to prove Theorem 6.1, however, we need to show that the expected number of iterations is only $O(\log^3 n)$.

Define $I^0 = \{i \in I | \mathrm{succ}(i) = \emptyset\}$.

PROPOSITION 6.14.

$$|I^0| + \sum_{i \in I \setminus I^0} \sum_{j \in \mathrm{succ}^{-1}(i)} 1/|\mathrm{succ}(j)| = |I|.$$

Proof. The sum is well defined since in each inner sum, $\{i\} \cup \mathrm{succ}(i) \subset \mathrm{succ}(j)$, and hence $|\mathrm{succ}(j)| \geq 2$. Note that

$$\sum_{i \in I \setminus I^0} \sum_{j \in \mathrm{succ}^{-1}(i)} 1/|\mathrm{succ}(j)| = \sum_{i \in I \setminus I^0} |\mathrm{succ}(i)|/|\mathrm{succ}(i)| = |I| - |I^0| .$$

Consider iterations of steps 2–5. For $k \geq 0$, denote by I_k, I_k^0, $\mathrm{succ}_k : I_k \to 2^{I_k}$, $\mathrm{pred}_k : I_k \to 2^{I_k}$, respectively, the contents of the sets I, I_k^0, and the corresponding succ and pred functions in the beginning of the kth iteration. For $k \geq 0$ and $i \in I_k$ we have $\mathrm{succ}_k(i) = \mathrm{succ}_0(i) \cap I_k$ and $\mathrm{pred}_k(i) = \mathrm{pred}_0(i)$ (since if $j \in \mathrm{pred}(i)$ is located, then so is i).

For $k \geq 0$, define

$$L_k = \left\{ i \in I_k \setminus I_k^0 | \sum_{j \in \mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \geq 1/4 \right\} .$$

PROPOSITION 6.15. *There exists a constant ρ_0 such that for every $k \geq 0$ and $i \in L_k \cup I_k^0$,* $\mathrm{Prob}\{i \notin I_{k+1}\} \geq \rho_0$.

Proof. Consider $i \in I_k^0$. It follows from Corollary 6.12 that if $i \in I'$ in some iteration of Steps 3–5, then i gets located. With probability at least 1/2, $i \in I'$ at the iteration where $\ell = 2^{\lfloor \log |I_k| \rfloor}$. Consider $i \in L_k$. For $0 \leq h \leq \lfloor \log |I_k| \rfloor$, denote

$$T_h = \{j \in \mathrm{succ}_k^{-1}(i) | 2^h \leq |I_k|/|\mathrm{succ}_k(j)| \leq 2^{h+1}\} .$$

It follows that

$$\sum_{0 \leq h \leq \lfloor \log |I_k| \rfloor} 2^h |T_h|/|I_k| \geq (1/2) \sum_{j \in \mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \geq 1/8 .$$

For $0 \leq h \leq \lfloor \log |I_k| \rfloor$, consider the iteration where $\ell = 2^h$. Consider an element $j \in T_h$. Note that $\{i\} \cup \mathrm{succ}_k(i) \subset \mathrm{succ}_k(j)$. Hence, $|\mathrm{succ}_k(j)| \geq 2$ and $\ell \leq |I_k|/|\mathrm{succ}_k(j)| \leq |I_k|/2$. Therefore, the probability that $\mathrm{succ}_k(j) \cap I' = \emptyset$ is at least

$$(1 - \ell/|I_k|)^{|\mathrm{succ}_k(j)|} \geq (1 - \ell/|I_k|)^{2|I_k|/\ell} \geq 1/16 .$$

Note that these probabilities are dependent for different elements of T_h. The probability that $j \in I'$ is at least $\ell/|I_k|$. These probabilities are independent for different elements of T_h. Using Corollary 6.12, it follows that the probability that i is not located in the iteration where $\ell = 2^h$ is at most

$$(1 - \ell/|I_k|)^{|T_h|} \leq ((1 - \ell/|I_k|)^{|I_k|/\ell})^{\ell |T_h|/|I_k|} \leq 4^{-2^h |T_h|/|I_k|} .$$

These probabilities are independent for different iterations. Hence, the probability that i is not located in any iteration is at most

$$\prod_{0\le h\le\lfloor\log|I_k|\rfloor} 4^{-2^h|T_h|/|I_k|} = 4^{-\sum_{0\le h\le\lfloor\log|I_k|\rfloor} 2^h|T_h|/|I_k|} \le 4^{-1/8} < 0.85\,.$$

Thus, i is located with probability at least 0.15.

PROPOSITION 6.16.

$$\sum_{i\in L_k}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \ge 3(|I_k|-|I_k^0|)/4\,.$$

Proof. The proof follows from Proposition 6.14 and from

$$\sum_{i\in I_k\setminus(I_k^0\cup L_k)}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \le (|I_k|-|I_k^0|-|L_k|)/4 \le (|I_k|-|I_k^0|)/4\,.$$

PROPOSITION 6.17. *Denote* $\rho_2=3\rho_0/8$ *and* $\rho_1=\rho_0/2$. *With probability at least* ρ_1,

$$|I_k^0\setminus I_{k+1}| + \sum_{i\in I_k\setminus(I_k^0\cup I_{k+1})}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \ge \rho_2|I_k|\,.$$

Proof. Consider the random variable

$$R = |I_k^0\setminus I_{k+1}| + \sum_{i\in L_k\setminus I_{k+1}}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)|\,.$$

Note that

$$\sum_{i\in I_k\setminus(I_k^0\cup I_{k+1})}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)| \ge \sum_{i\in L_k\setminus I_{k+1}}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)|\,.$$

Hence, it suffices to show that $\mathrm{Prob}\{R\ge\rho_2|I_k|\}\ge\rho_1$. It follows from Proposition 6.15 that the expected value of R is

$$E(R) \ge \rho_0|I_k^0| + \rho_0\sum_{i\in L_k}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)|\,.$$

The maximum value of R is at most

$$R_{\max} = |I_k^0| + \sum_{i\in L_k}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)|\,.$$

Note that

$$E(R) \le \mathrm{Prob}\{R<E(R)/2\}E(R)/2 + \mathrm{Prob}\{R\ge E(R)/2\}R_{\max}\,.$$

Therefore,

$$\mathrm{Prob}\{R\ge E(R)/2\} \ge E(R)/(2R_{\max}-E(R)) \ge E(R)/(2R_{\max}) \ge \rho_0/2\,.$$

It follows that with probability at least $\rho_0/2$,

$$R \ge \left(\rho_0|I_k^0| + \rho_0\sum_{i\in L_k}\sum_{j\in\mathrm{succ}_k^{-1}(i)} 1/|\mathrm{succ}_k(j)|\right)/2 \ge (\rho_0|I_k^0|+3\rho_o(|I_k|-|I_k^0|)/4)/2 \ge 3\rho_0|I_k|/8\,.$$

The second to last inequality follows from Proposition 6.16.

PROPOSITION 6.18. *Consider a sequence of* $h' \geq 4c_0/\rho_1\lceil\log_{c_1}|I_k|\rceil$ *iterations of steps* 2–5, *starting at the kth iteration, where* $c_0 \geq 1$ *and* $c_1 > 1$ *are constants that are specified below. With probability at least* $1/2$, $|I_{k+h'}| < (1-\rho_2/2)|I_k|$.

Proof. Let H be the set of all $1 \leq h \leq h'$ such that

$$|I^0_{k+h-1} \setminus I_{k+h}| + \sum_{i\in I_{k+h-1}\setminus(I^0_{k+h-1}\cup I_{k+h})}\ \sum_{j\in \mathrm{succ}^{-1}_{k+h-1}(i)} 1/|\mathrm{succ}_{k+h-1}(j)| \geq \rho_2|I_{k+h-1}|\,.$$

It follows from Proposition 6.17 that for $1 \leq h \leq h'$, $h \in H$ with probability at least ρ_1. Hence, using bounds on the binomial distribution, with probability $1-\exp(O(-h'\rho_1)) \geq 1/2$, $|H| \geq c_0\lceil\log_{c_1}|I_k|\rceil$. We assume that $|H| \geq c_0\lceil\log_{c_1}|I_k|\rceil$ and $|I_{k+h'}| \geq (1-\rho_2/2)|I_k|$ and obtain a contradicition. Consider $h \in H$. It follows, using Proposition 6.14, that

$$\begin{aligned}&|I^0_{k+h-1} \cap I_{k+h}| + \sum_{i\in I_{k+h}\setminus I^0_{k+h-1}}\ \sum_{j\in \mathrm{succ}^{-1}_{k+h-1}(i)} 1/|\mathrm{succ}_{k+h-1}(j)|\\ &= |I^0_{k+h-1} \cap I_{k+h}| + \textstyle\sum_{i\in I_{k+h}\setminus I^0_{k+h-1}} |\mathrm{succ}_{k+h}(i)|/|\mathrm{succ}_{k+h-1}(i)| < (1-\rho_2)|I_{k+h-1}|.\end{aligned}$$

We obtain that

$$\begin{aligned}\sum_{i\in I_{k+h}\setminus I^0_{k+h-1}} |\mathrm{succ}_{k+h}(i)|/|\mathrm{succ}_{k+h-1}(i)| &< (1-\rho_2)|I_{k+h-1}| - |I^0_{k+h-1}\cap I_{k+h}|\\ &\leq (1-\rho_2)|I_{k+h}|/(1-\rho_2/2) - |I^0_{k+h-1}\cap I_{k+h}|\,.\end{aligned}$$

Denote $\rho_3 = (1-\rho_2)/(1-\rho_2/2)$ and $\rho_4 = (1+\rho_3)/2$. Note that $\rho_3 < 1$ and $\rho_3 < \rho_4 < 1$. The number of elements in the summation that are not smaller than ρ_4 is less than

$$(\rho_3|I_{k+h}| - |I^0_{k+h-1}\cap I_{k+h}|)/\rho_4\,.$$

The number of elements in the summation is $|I_{k+h}\setminus I^0_{k+h-1}| = |I_{k+h}| - |I^0_{k+h-1}\cap I_{k+h}|$. Hence, more than

$$(1-\rho_3/\rho_4)|I_{k+h}| + (1/\rho_4 - 1)|I^0_{k+h-1}\cap I_{k+h}| \geq (1-\rho_3/\rho_4)|I_{k+h}|$$

elements are smaller than ρ_4. Using $|I_{k+h}| \geq (1-\rho_2/2)|I_k|$, it follows that after each iteration in H, for more than $(1-\rho_3/\rho_4)(1-\rho_2/2)|I_k|$ elements, the number of successors (size of succ_{k+h}) reduces by a factor of at least ρ_4. Let

$$c_0 = 1/((1-\rho_3/\rho_4)(1-\rho_2/2)) + 1$$

and $c_1 = (1-\rho_4)^{-1}$. The size of elements in the range of succ_k is bounded by I_k. Hence

$$|H| < c_0\lceil\log_{c_1}|I_k|\rceil\,,$$

which contradicts our assumption.

COROLLARY 6.19. *The expected number of iterations of steps* 2–5 *is* $O(\log^2 n)$.

Proof of Theorem 6.1. Algorithm 6.8 terminates with a correct solution. Each execution of Steps 2–5 amounts to $O(\log n)$ calls to Algorithms 6.3 and 6.6. Hence it follows from Corollary 6.19 that Algorithm 6.8 performs an expected number of $O(\log^3 n)$ applications of Algorithms 6.3 and 6.6. Recall that Algorithms 6.3 and 6.6 run in $O(mn)$ time sequentially, and on a CRCW PRAM, in $O(n)$ time using $O(m)$ processors. It follows that the expected running time of Algorithm 6.8 is as stated in the theorem. □

7. Monotone systems. Consider the following two operations on vectors in R^n:

$$\boldsymbol{x} \vee \boldsymbol{y} = (\max\{x_1, y_1\}, \dots, \max\{x_n, y_n\})^T \quad \text{(the } \textit{join} \text{ of } \boldsymbol{x} \text{ and } \boldsymbol{y}\text{)},$$
$$\boldsymbol{x} \wedge \boldsymbol{y} = (\min\{x_1, y_1\}, \dots, \min\{x_n, y_n\})^T \quad \text{(the } \textit{meet} \text{ of } \boldsymbol{x} \text{ and } \boldsymbol{y}\text{)}.$$

Consider a TVPI system as in Definition 1.1, where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is a *pre-Leontief* matrix, that is, each row of $\boldsymbol{A}$ contains at most one positive (resp., negative) entry. The set of solutions of such a system constitutes a semilattice relative to the $\vee$ (resp., $\wedge$) operation (see Cottle and Veinott [10]). It follows that if all the variables are bounded from above (resp., below), there exists a solution where all the variables are maximized (resp., minimized) simultaneously, hence, the vector $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) is feasible. Also, the associated graph of such a system (see Definition 2.1) does not contain edges from $\underline{V}$ to $\overline{V}$ (resp., from $\overline{V}$ to $\underline{V}$).

A TVPI system where the two nonzero coefficients in each inequality have opposite signs (equivalently, both $\boldsymbol{A}^T$ and $-\boldsymbol{A}^T$ are pre-Leontief) is called *monotone*. The set of solutions of monotone systems constitutes a lattice relative to the $\vee$ and $\wedge$ operations. Note that the converse is also true: If a polyhedral set constitutes a lattice relative to the $\vee$ and $\wedge$ operations, then it can be represented as the set of solutions of some monotone system. This converse is a direct consequence of the following result due to Veinott [18]: a polyhedral set is a semilattice relative to the $\vee$ (resp., $\wedge$) operation if and only if it constitutes the set of solutions of some linear system $\boldsymbol{Ax} \leq \boldsymbol{b}$ (resp., $\boldsymbol{Ax} \geq \boldsymbol{b}$), where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is pre-Leontief. Consider, for example, the nonmonotone system in Fig. 6. The points (5, 1) and (3, 4) are feasible, but the point $(5, 1) \wedge (3, 4) = (3, 1)$ is not. Monotone systems are related to generalized network flow problems: The linear programming dual of the uncapacitated generalized transshipment problem is monotone. In [4] and [7] we present algorithms for generalized network flow problem that exploit this relationship.

$$\begin{array}{rcrl} y & - & 3x \leq & 0 \\ 8y & - & 3x \leq & 42 \\ -y & + & x \leq & 1 \\ -y & + & x \leq & -3 \end{array} \qquad \begin{array}{rcrl} 3y & - & 4x \leq & 0 \\ 5y & + & x \leq & 46 \\ -y & + & x \leq & 4 \\ -2y & + & 3x \leq & -7 \end{array}$$

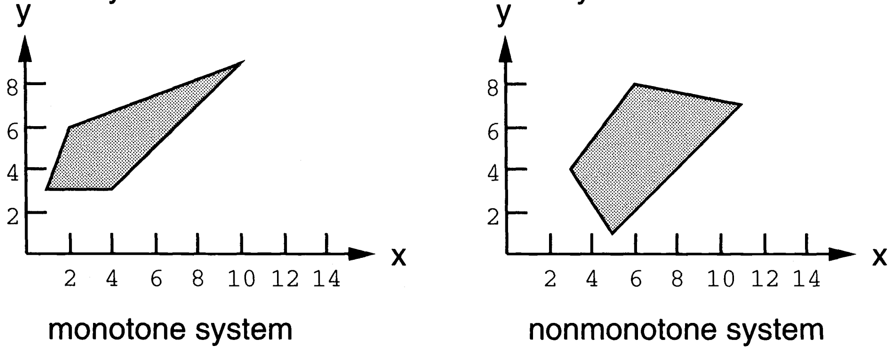


FIG. 6. *Examples of a monotone system and a nonmonotone system.*

The algorithms presented here for solving the feasibility of TVPI systems can be adapted to find the vectors $\boldsymbol{x}^{\min}$, $\boldsymbol{x}^{\max}$ for monotone systems. Note that in the associated graph of a monotone system there are no edges passing between the two sets of vertices $\overline{V}$ and $\underline{V}$. It follows that all certificates are either closed certificates or simple paths.

We consider an application of Algorithm 3.3 to a monotone system, with the goal of finding the point $\boldsymbol{x}^{\max}$. It suffices to rely on a simpler notion of locating values: *Locating* the value ξ means locating it with respect to $x_i^{\max}$ (rather than determining its position relative to $x_i^{\min}$ as well). Problem 3.6 amounts to the following.

PROBLEM 7.1 [*locate a pool of values*]. *Given are values ξ_i $(i \in I)$ for the corresponding variable x_i $(i \in I)$. Locate these values, i.e., determine if $\xi_i > x_i^{\max}$ $(i \in I)$. Choose intervals J_i $(i \in I)$ as follows. $J_i = [\xi_i, \infty]$, if $\xi_i \leq x_i^{\max}$ and $J_i = [-\infty, \xi_i]$, if $\xi_i \geq x_i^{\max}$ $(i \in I)$.*

It is easy to see that the intervals J_i are such that $\{x | \bigwedge_{i \in I} x_i \in J_i\}$ contains the simultaneous maximum.

Consider step 7 of Algorithm 3.3. The interval S_i' are feasible for x_i $(1 \leq i \leq n)$. Hence, for $1 \leq i \leq n, b_i' = x_i^{\max}$ (x_i is unbounded if and only if $b_i' = \infty$). If all variables are bounded, $\boldsymbol{b}' = \boldsymbol{x}^{\max}$. Otherwise, the algorithm supplies dependencies that allow us to compute feasible solutions when we determine some variables and maximize others.

REMARK 7.2. *An algorithm that computes $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) for feasible monotone systems can be applied to compute $\boldsymbol{x}^{\max}$ (resp., $\boldsymbol{x}^{\min}$) for* TVPI *systems where the matrix $\boldsymbol{A}^T$ (resp., $-\boldsymbol{A}^T$) is pre-Leontief. Consider a system $\boldsymbol{Ax} \leq \boldsymbol{b}$ where at most one of the two nonzero coefficients in each row is positive. Obtain a monotone system $\boldsymbol{A}'\boldsymbol{x} \leq \boldsymbol{b}'$ by omitting all rows of $\boldsymbol{A}$ where there are two negative entries. Compute the vector $\boldsymbol{y}$ that maximizes all variables in the resulting monotone system. If the original system $\boldsymbol{Ax} \leq \boldsymbol{b}$ is feasible, $\boldsymbol{y}$ is the vector that maximizes all variables.*

8. Concluding remarks. In this paper we considered systems of linear inequalities, where each inequality has at most two nonzero coefficients (TVPI systems). We presented algorithms that solve the feasibility problem, that is, either find a point that satisfies all the inequalities or conclude that no such point exists. We gave a $\tilde{O}(mn^2)$ deterministic algorithm and a $\tilde{O}(n^3 + mn)$ expected time randomized algorithm. The complexities of the respective parallel implementations are: $\tilde{O}(n)$ time using $O(mn)$ processors, and $\tilde{O}(n)$ expected time using $O(n^2+m)$ processors. Although the analysis of these algorithms seems quite lengthy, the algorithms themselves are simple. The underlying computation amounts to the basic Bellman-Ford and Floyd-Warshall [8] shortest-path algorithms where only simple data structures are used. Our algorithms are strongly polynomial, that is, the number of operations does not depend on the size of input numbers. We comment that the size (number of bits) of the numbers generated by the algorithms is $O(nC)$, where C is the size of the largest coefficient in the input.

We give some comments and suggestions for further research:

The time complexity of the randomized algorithm involves many logarithmic factors ($\log^5 n$). We suspect that a more careful analysis may eliminate some of these factors. The constant factors in the algorithms of §5 are quite large, but probably can be reduced significantly. We believe that some appropriate version of the algorithms presented here could work well in practice. Another obvious question is whether the $\tilde{O}(n^3 + mn)$ bound can be achieved deterministically.

We discuss the possibility of improving over a $\tilde{O}(n^3 + mn)$ bound. The $\tilde{O}(n^3)$ factor results from the all-pairs shortest-paths Floyd-Warshall computation and is inherent from our basic framework. A different approach, however, may yield a $\tilde{O}(mn)$ algorithm. We note that more work efficient parallel all-pairs shortest-paths algorithms that are applicable for semirings, can be combined with our techniques (instead of the Floyd-Warshall algorithm) to obtain better bounds. In particular, a polylogarithmic time $f(m, n) = o(n^3)$ work all-pairs shortest-paths algorithm yields a $\tilde{O}(f(m, n) + mn)$ algorithm for solving TVPI systems. We comment that better bounds are obtained for TVPI systems where the associated graph has some special structure for which more work-efficient shortest-paths algorithms exist. When the associated graph has a separator decomposition using $O(n^\mu)$ size separators (e.g., planar graphs, k-dimensional grid graphs bounded on most dimensions, or bounded tree-width graphs), the corresponding TVPI systems can be solved in $\tilde{O}(n^{3\mu} + mn)$ time, using a shortest paths algorithm by the first author [5]. Consider, as an example, a TVPI system such that the variables correspond to vertices of a three-dimensional grid, and all inequalities are "local" (that is, for each inequality, the distance on the grid between the two vertices corresponding to the variables in the inequality is bounded by some constant). The associated graph has a separator decomposition using $O(n^{2/3})$ size separators, and hence, these TVPI systems can be solved in $\tilde{O}(mn)$ expected time.

The feasibility problem of monotone systems includes as a very special case the problem of detecting existence of negative weight directed cycles in a graph with n nodes, m edges, and real weights associated with the edges (using a well-known reduction). The best known bound for detecting negative weight cycles is $O(mn)$ and hence we believe it is unlikely that a $\tilde{o}(mn)$ algorithm exists for solving TVPI systems. We sketch the reduction. Consider a weighted graph $G = (V, E, w)$, where $w : E \to R$. The corresponding monotone system is as follows. For each node $v \in V$ assign a variable x_v. For each edge $e = (u, v) \in E$ assign the inequality $x_v - x_u \leq w(e)$. It follows from Proposition 2.10 that G contains a negative weight cycle if and only if the system is infeasible.

Our algorithms solve the feasibility problem of TVPI systems. They can be adapted, however, to (i) find a solution that maximizes a specific variable (ii) and find the lexicographic maximum. Questions that remain open regard finding an optimal solution relative to an arbitrary linear objective function. It is not known whether a strongly polynomial time algorithm exists for the problem, or whether we can achieve a time bound that is better than specializations of existing general LP algorithms. A partial result is due to Cosares [9], who showed that when the objective function has a fixed number of nonzero entries, the problem can be solved in strongly polynomial time bounds. The degree of the polynomial, however, grows linearly with the number of nonzero entries.

We discuss the parallel complexity of solving TVPI systems. Lueker, Megiddo, and Ramachandran [14] showed that the problem of finding an optimal solution relative to a general objective function is $\mathcal{P}$ complete. It is not known, however, whether the feasibility problem is $\mathcal{P}$ complete. The algorithms presented in this paper have a parallel running time of $\tilde{O}(n)$. This is the best known parallel time bound achievable by a polynomial number of processors. Lueker, Megiddo, and Ramachandran [14] gave an algorithm for the problem that runs in polylogarithmic time, but uses $n^{O(\log n)}$ processors.

Acknowledgments. The authors would like to thank the anonymous referee for very careful reading and many good suggestions for improving the presentation.

REFERENCES

[1] I. Adler and S. Cosares, *A strongly polynomial algorithm for a special class of linear programs*, Oper. Res., 39 (1991), pp. 955–960.

[2] B. Aspvall, *Efficient algorithms for certain satisfiability and linear programming problems*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1980.

[3] B. Aspvall and Y. Shiloach, *A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality*, SIAM J. Comput., 9 (1980), pp. 827–845.

[4] E. Cohen, *Combinatorial Algorithms for Optimization Problems*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1991.

[5] ———, *Efficient parallel shortest-paths in digraphs with a separator decomposition*, in Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, 1993, pp. 57–67.

[6] E. Cohen and N. Megiddo, *Improved algorithms for linear inequalities with two variables per inequality*, in Proc. 23rd Annual ACM Symposium on Theory of Computing, ACM Press, New York, 1991, pp. 145–155. SIAM J. Comput., this paper.

[7] E. Cohen and N. Megiddo, *New algorithms for generalized network flows*, in Proc. of the 1st Israeli Symposium on the Theory of Computing and Systems, D. Dolev, Z. Galil, and M. Rodeh, eds., Lecture Notes in Comput. Sci., 601, Springer-Verlag, New York, 1992, pp. 103–114.

[8] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[9] S. Cosares, *On the complexity of some primal-dual linear programming pairs*, Ph.D. thesis, Department of IEOR, University of California, Berkeley, CA, 1988.

[10] R. W. Cottle and A. F. Veinott Jr., *Polyhedral sets having a least element*, Math. Prog., 3 (1972), pp. 238–249.

[11] W. Feller, *An introduction to probability theory and its applications*, John Wiley and Sons, New York, 1950.

[12] D. Hochbaum and J. Naor, *Simple and fast algorithms for linear and integer programs with two variables per inequality*, Proceedings of the 2nd MPS Conference on Integer Programming and Combinatorial Optimization (IPCO), CMV, (1992), pp. 44–59; SIAM J. Computing, this issue, pp. 1179–1192.

[13] R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, vol. A, J. van Leeuwenm, ed., MIT Press/Elsevier, Cambridge, MA, 1991, pp. 869–941.

[14] G. S. Lueker, N. Megiddo, and V. Ramachandran, *Linear programming with two variables per inequality in poly log time*, SIAM J. Comput., 19 (1990), pp. 1000–1010.

[15] N. Megiddo, *Towards a genuinely polynomial algorithm for linear programming*, SIAM J. Comput., 12 (1983), pp. 347–353.

[16] C. G. Nelson, *An* $n^{O(\log n)}$ *algorithm for the two-variable-per-constraint linear programming satisfiability problem*, Tech. Report AIM-319, Stanford University, Standord, CA, 1978.

[17] R. Shostak, *Deciding linear inequalities by computing loop residues*, J. Assoc. Comput. Mach., 28 (1981), pp. 769–779.

[18] A. F. Veinott Jr., *Representation of general and polyhedral subsemilattices and sublattices of product spaces*, Linear Algebra Appl., 114/115 (1989), pp. 681–704.